

# ServerlessLoRA: Minimizing Latency and Cost in Serverless Inference for LoRA-Based LLMs

Yifan Sui\*  
Shanghai Jiao Tong University  
Shanghai, China  
suiyifan@sjtu.edu.cn

Hao Wang  
Stevens Institute of Technology  
Hoboken, USA  
hwang212@stevens.edu

Hanfei Yu  
Stevens Institute of Technology  
Hoboken, USA  
hyu42@stevens.edu

Yitao Hu  
Tianjin University  
Tianjin, China  
yitao@tju.edu.cn

Jianxun Li†  
Shanghai Jiao Tong University  
Shanghai, China  
lijx@sjtu.edu.cn

Hao Wang  
Stevens Institute of Technology  
Hoboken, USA  
hwang9@stevens.edu

## 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 can effectively serve general LLM but fail with 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 *ServerlessLoRA*, a novel serverless inference system designed for faster and cheaper LoRA LLM serving. *ServerlessLoRA* 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, *ServerlessLoRA* employs contention aware batching and offloading to mitigate GPU resource conflicts during bursty workloads. Experiment on industrial workloads demonstrates that *ServerlessLoRA* 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 [39] and Qwen [32], 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 must host many versions of LLMs for different users and tasks, which results in massive GPU and monetary costs when maintaining all LLMs in long-running GPU instances.

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 [5], NVIDIA DGX [11], and ServerlessLLM [16].

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.** Although many LoRA functions are fine-tuned atop the same backbone LLM<sup>1</sup>, current serverless frameworks treat each function as an independent unit—even though 99% of the weights are identical across functions. This redundancy leads to repeated loading of the same backbone, resulting in unnecessary GPU wastage and high loading latency. **Observation 2: LoRA introduces costly artifacts loading latency.** Existing solutions [16, 25, 44] focus solely on loading the backbone checkpoint (from RAM, SSD, or remote storage) while neglecting the overhead of loading libraries, LoRA adapters, and just-in-time (JIT) compiled CUDA kernels. **Observation 3: Serving multi-LoRA magnifies resource contention.** LoRA-based applications often utilize diverse backbone models and adapters with varying mappings [27, 48], necessitating the invocation of multiple LoRA functions. The concurrent loading of artifacts and inference exacerbates resource contention, thereby greatly increasing Time-To-First-Token (TTFT).

\*This work was performed when Yifan Sui was a remote intern student advised by Dr. Hao Wang at the IntelliSys Lab of Stevens Institute of Technology.

†Corresponding author

<sup>1</sup>We use “backbone” to refer to the base LLM.

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 several key challenges. First, how can we share the backbone LLM while guaranteeing each function’s isolation and security? Although some solutions [7, 35, 40] allow backbone sharing among adapters, they typically run all adapters and backbones within a single process, thereby violating the fundamental isolation paradigm of serverless computing. Second, given limited GPU resources, it is infeasible to pre-load all functions simultaneously; thus, minimizing TTFT without consuming additional GPU resources remains a critical challenge. Third, how can we serve multiple LoRA functions on demand without incurring GPU contention or excessive LoRA switch latency?

To address these challenges, We present *ServerlessLoRA*, a cost-efficient serverless inference system designed for LoRA. First, *ServerlessLoRA* enables backbone LLM sharing across isolated functions, thereby avoiding repeated loading while preserving the serverless security principle. Second, we propose a comprehensive pre-loading approach that goes beyond traditional LLM pre-fetching—which only transfers raw data from remote storage to DRAM—by covering every preparatory step for LoRA inference: loading artifacts into memory, initializing required libraries and the CUDA context, and JIT-compiling necessary CUDA kernels. Third, to support various backbones and LoRA adapters without resource contention, we propose contention-aware batching that prevents conflicts when multiple LoRA functions share GPUs, along with dynamic off-loading that continuously monitors LoRA usage and evicts idle artifacts to free GPU memory during bursty workload.

We summarize *ServerlessLoRA*’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 *ServerlessLoRA*, a novel serverless framework that makes LoRA inference faster and cheaper. Leveraging backbone sharing and extended pre-loading of LoRA artifacts, *ServerlessLoRA* substantially reduces GPU consumption and cold-start latency.
- We thoroughly evaluate *ServerlessLoRA* on industrial workloads and benchmark LLMs. Extensive experiments show reductions of up to 86% TTFT and 89% monetary cost, compared to state-of-the-art solutions.

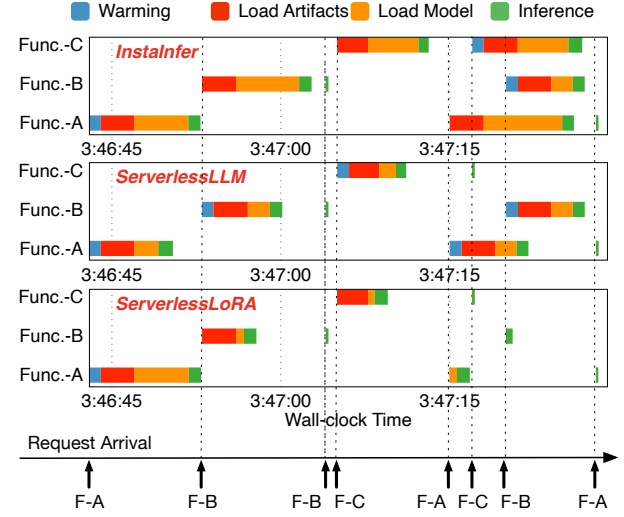


Figure 1. Time breakdown of LoRA functions’ invocations.

## 2 Background and Motivation

### 2.1 Serverless LLM Inference

Serverless computing has emerged as a compelling paradigm for deploying LLM inference services due to its proven effectiveness in real-world deployments. Major cloud providers like AWS and NVIDIA have successfully integrated serverless architectures for LLM services [5, 11], demonstrating the potential of serverless for serving LLMs at scale.

The serverless paradigm offers significant advantages for LLM inference. First, it provides superior GPU and cost efficiency compared to serverful<sup>2</sup> LLM deployments such as vLLM [21]. As the request arrival is variable, maintaining continuously running GPU instances causes huge GPU consumption. Our evaluation result in Fig. 2a demonstrates that serverless inference offers substantially higher cost-effectiveness over serverful solutions due to the pay-per-use pricing, which eliminates idle charges. When serving a Llama2-7B LLM using Azure Function Trace [34], serverless solutions (ServerlessLLM [16] and InstaInfer [38]) demonstrate 3× higher cost-effectiveness<sup>3</sup> compared to serverful solutions (vLLM [21] and dLoRA [40]).

Furthermore, serverless computing delivers superior elasticity and scalability, meeting the critical requirements for LLM inference workloads. Azure LLM inference traces [37] reveal that request loads fluctuate dramatically, with peak loads reaching up to 34.6× higher than valley periods. Unlike serverful solutions that typically scale at the minute level, serverless platforms can respond within seconds to sudden request bursts. This rapid scaling capability ensures consistent performance even under highly variable workload conditions.

<sup>2</sup>We use “serverful” to refer to traditional VM-based long-running serving.

<sup>3</sup>Defined as  $1/(E2E\_latency \times Monetary\_Cost)$

## 2.2 Limitations of Serverless on LoRA Inference

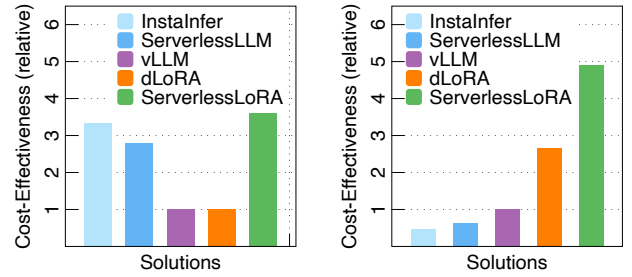
While serverless computing offers advantages for general LLM inference, real-world applications rarely rely on general-purpose models. Organizations typically deploy multiple specialized models fine-tuned for specific domains, tasks, or user segments. Low-Rank Adaptation (LoRA) has become the predominant fine-tuning method due to its parameter efficiency, allowing practitioners to create specialized models without retraining the entire backbone LLM. Due to the decoupling of backbone and LoRA parameters, the inference can be operated by separately calculating the attention of the backbone and LoRA adapter, and finally gathering their results as the final output.

However, our observations reveal a critical inefficiency in current serverless solutions when serving LoRA LLMs. When serving multiple LoRA functions, existing serverless solutions require each function to load the entire backbone LLM independently, despite these functions sharing the same underlying model. This creates massive redundancy.

This redundancy manifests in excessive GPU consumption and elevated monetary costs. Since even a 7B parameter LLM consumes an entire NVIDIA A10 GPU during inference, and the backbone accounts for up to 99% of an LLM’s parameters, this redundancy results in substantial wasted GPU resources. The problem is further exacerbated by serverless platforms’ keep-alive policies, which maintain each invoked function for several minutes after execution to mitigate cold starts on subsequent requests. In the LoRA scenario, this means each function’s complete LLM occupies expensive GPU resources during the whole idle periods, multiplying costs unnecessarily. Given that GPU costs constitute approximately 90% of an invocation’s total monetary expense<sup>4</sup>, 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, due to the massive redundancy, serverless solutions’ cost-effectiveness decreases significantly.

Although approaches like Punica [7], S-LoRA [35], and dLoRA [40] enable backbone sharing across LoRA adapters, they execute all adapters and backbones inside the same OS process and one CUDA context. Consequently, all runtime artifacts become globally visible: the key-value attention cache (KV cache), CUDA kernels and GPU streams, memory allocator arenas, and scheduler internal queues all share the same address space. Such designs directly contradict the isolation requirement and security principle of serverless computing, which requires strict isolation between functions.

Furthermore, current serverless approaches impose heavy startup delays for LoRA functions. Loading a backbone LLM with billions of parameters can take tens of seconds per function. Since each function independently loads the identical backbone, the aggregate cold start latency increases linearly



(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).

with the number of LoRA functions. Beyond the backbone loading, current serverless inference overlooks the 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.

To comprehensively visualize the effect of backbone redundancy and LoRA artifacts loading, we present a time breakdown of each request’s E2E latency running the Azure Trace for three Llama2-13B LoRA functions in Fig. 1. Both InstaInfer and ServerlessLLM introduce significant cold-start latency due to repeatedly loading the backbone and other LoRA artifacts. Although ServerlessLLM, the state-of-the-art serverless LLM inference solution, has effectively reduced backbone loading latency. However, it still experiences significant cold-start latency.

## 2.3 The Necessity of 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-based 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. *Second*, as Fig. 1 shows, loading LLM artifacts—such as model weights, libraries, CUDA contexts, and compiled kernels—accounts for over 90% of the startup time. This observation motivates our second approach: pre-loading these artifacts prior to function invocation to reduce startup latency and improve responsiveness under bursty workloads.

## 2.4 Opportunity of Backbone Sharing and Pre-Loading

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,

<sup>4</sup>Based on the Alibaba Cloud serverless pricing model [9].

indicating many functions can share the same backbone. Second, attention calculation for the backbone and LoRA adapter can be performed separately, ensuring that the sharing does not compromise the inference accuracy of individual functions.

Serverless platforms’ function keep-alive policies create a unique opportunity for pre-loading without additional resources. Numerous containers and GPUs remain idle after servicing requests, maintained for future invocations. Besides, to deal with peak workload, functions are usually over-allocated with sufficient memory [14, 17, 33, 34, 43, 47, 50]. Thus, the vast memory gap between running and idle states presents the opportunity for pre-loading without extra wastage.

### 3 System Overview

#### 3.1 Goals & Challenges

*ServerlessLoRA* 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, *ServerlessLoRA* should minimize both to lower overall cost.

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

To meet our goals, we face 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 *ServerlessLoRA*’s System Architecture

We introduce the architecture of *ServerlessLoRA*, a model-sharing based serverless LLM inference system aiming at

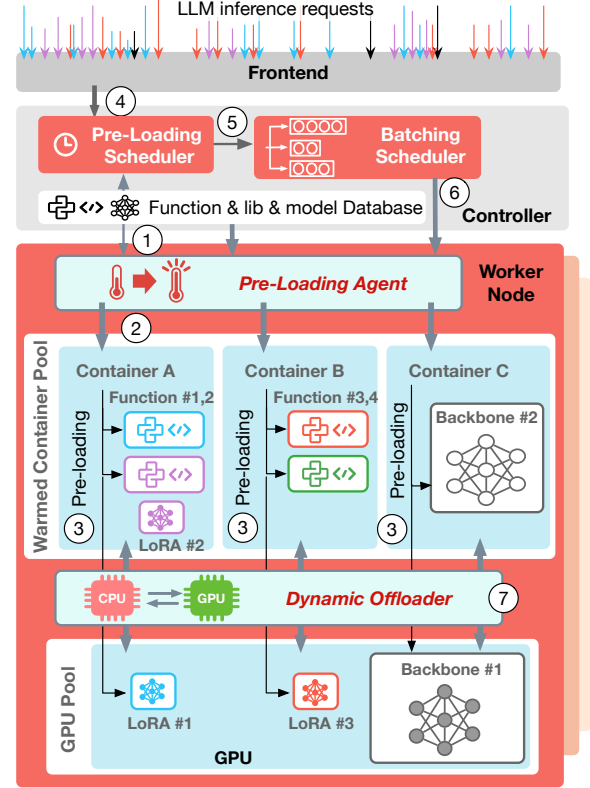


Figure 3. System overview.

minimizing startup latency and achieving high scalability under minimal resource and monetary cost. We design a secure LLM sharing mechanism that allows functions share backbone LLM safely. To ensure the model sharing mechanism can achieve the above goals, *ServerlessLoRA* 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 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 get the optimal pre-loading decision under the limited resources.

**Batching Scheduler** determines the batch size and queuing time of each function. It aims at fully utilizing GPU resource 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 corresponding command to each container and GPU. Besides, it manages the pre-warming and keep-alive of function instances.

**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 remove them, until there is enough space to serve all requests.



### 3.3 ServerlessLoRA’s Workflow

The workflow of *ServerlessLoRA* can be divided into two main stages: the pre-loading stage and the request serving stage.

The pre-loading stage contains three steps:

**Making Pre-Loading decisions (Step 1):** The Pre-Loading Scheduler analyzes the available resources and each function’s request arrival frequency to determine the optimal pre-loading decision. It decides which function should be fully loaded in GPU memory for minimal startup and which function should be pre-loaded in container memory for sub-optimal acceleration.

**Command Dispatching (Step 2):** Once the pre-loading decision is made, the Pre-Loading Scheduler passes it to each worker node’s Pre-Loading Agent. The Agent then prepares corresponding containers and GPUs for pre-loading.

**Artifact Loading (Step 3):** The containers receive these commands and proceed to load libraries, models, CUDA kernels into container and GPU memory.

The request serving stage contains another four steps:

**Instance Selection (Step 4):** When a new request arrives, the Pre-Loading Scheduler selects a function instance that has the optimal pre-loaded components.

**Request Batching (Step 5):** The Batching Scheduler continuously gathering the incoming request stream and. Each function has its own batch queue and batch size.

**Dispatching Batched Requests (Step 6):** When the batch size is reached or the batching timeout occurs, the batched requests are then dispatched to the chosen function instance.

**Dynamic GPU Memory Management (Step 7):** Concurrently, the Dynamic Offloader examines the selected GPU’s available capacity. When the GPU memory nears saturation, it triggers a dynamic offloading operation, moving less frequently used models to container memory or clear their CUDA contexts.

## 4 System 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 [14, 17, 33, 34, 43, 47, 50]. Thus, we share the container among multiple functions in the pre-loading stage.

*ServerlessLoRA* 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 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:

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

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

The pre-loading decisions are guided by several constraint categories:

**Capacity constraints** limit memory consumption of pre-loaded artifacts within container and GPU memory.

**Assignment and precedence constraints** enforce loading order: models require libraries first, and CUDA kernels require models on GPU first.

**Backbone-adapter coupling constraints** ensure adapters are loaded in containers connected to the same GPU hosting their backbone, requiring adapters and backbones to share GPU placement

As PCKP has been proven to be NP-hard [30], while dynamic programming could theoretically yield optimal solutions, it is impractical for serverless environments. It requires  $O(2^{|F|} \cdot (|C| + |G|))$  time complexity, making it computationally infeasible for typical serverless deployments with numerous functions and instances, violating the milliseconds scheduling latency requirement of serverless platforms.

Thus, we implement a greedy algorithm that prioritizes artifacts based on their value density  $\rho_i^f = \frac{v_i^f}{w_i^f}$ .

The algorithm sorts all artifacts by value density and pre-loads them in order while respecting all constraints. This approach has a time complexity of  $O(|F|^2 \cdot (|C| + |G|))$ , making it practical for large-scale deployments while achieving near-optimal performance.

### 4.2 Adaptive Batching

To improve throughput and fully leverage pre-loaded artifacts to reduce cold starts, *ServerlessLoRA* batches requests into a pre-loaded function instance. Consequently, the acceleration effect of a LLM artifact can be used for multiple

requests, avoiding creating new instances and thereby further reducing TTFT.

To achieve the above goal, the Batching Scheduler needs to carefully determine the batch size and batch delay of each function. Too small batch size leads to insufficient utilization of pre-loaded artifacts. In contrast, as larger batch size results in larger computational load, too large batch size leads to high inference time cost and violates SLO.

Furthermore, as *ServerlessLoRA* supports running multiple functions by sharing the backbone LLM, to avoid GPU resource contention (which can lead to SLO violations when multiple functions run concurrently), the Batching Scheduler should consider resource contention among functions.

We formulate a two-layer batching approach that balances throughput maximization with SLO compliance.

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

$$T_i(b) = T_{0,i} + \alpha_i(b - 1), \quad (2)$$

where  $T_{0,i}$  represents the base inference time (pre-filling) for the first token, and  $\alpha_i$  denotes the marginal cost of adding each subsequent request to the batch. Therefore, through offline profiling, we can get the maximum batch size  $B_i$  within the SLO. Meanwhile, the system calculates the maximum batch delay based on current batch number  $N_i$ :

$$d_i = SLO_i - T_i(N_i) \quad (3)$$

The batch stops either  $N_i$  requests are collected or the batching delay  $d_i$  expires. With this design, under the premise of guaranteeing SLO, the Batch Scheduler tends to wait longer when the batch size is small, thereby collecting more future requests to fully utilize the pre-loaded artifacts.

At the global level, the Batching Scheduler addresses resource contention that occurs when multiple batches compete for the same GPU resources. When  $M$  batches are processed concurrently on a shared GPU, the inference time for each function's batch expands to:

$$T_i^{\text{eff}}(b) = M \cdot T_i(b) \quad (4)$$

To manage this contention, the Batching Scheduler dynamically prioritizes batches based on their *deadline margin*:

$$\Delta_i = SLO_i - (w_i + M \cdot T_i(b)) \quad (5)$$

where  $w_i$  represents the time already spent waiting. Batches with smaller deadline margins are prioritized for immediate processing, while those with larger margins can afford to wait longer and potentially accommodate more requests.

### 4.3 Dynamic GPU Offloading

With adaptive batching, each function instance is designed to handle up to a maximum batch size of concurrent requests while maintaining minimal TTFT. However, reaching this

maximum batch size requires sufficient GPU memory, as each request consumes memory for its KV cache.

A significant challenge arises because pre-loaded artifacts, while beneficial for accelerating TTFT, consume non-negligible GPU memory even when their corresponding functions are non-invoked. During bursty periods, where functions need to serve numerous requests concurrently so that requests can quickly fill up the batch queue, these unrelated artifacts effectively reduce the available memory that could otherwise store KV caches for arriving requests. This memory contention prevents invoked functions from reaching their maximum batch size.

Therefore, to guarantee that the invoked functions have enough memory to reach the maximum batch size, we propose the Dynamic Offloader to offload unrelated LLM artifacts from GPUs and release resources for future requests.

**Offloading Policy Design.** As offloading pre-loaded artifacts reduces the potential acceleration for corresponding functions' future requests, we design an offloading policy that removes the least-valuable artifacts while preserving the overall potential acceleration of other functions.

The fundamental goal of our offloading policy is to minimize performance degradation when freeing GPU memory. This naturally leads us to formulate the problem as a value-optimization challenge. When a new request requires  $Q_g$  additional memory on GPU  $g$ , sufficient memory needs to be freed:

$$\sum_{f \in F} [w_{MG}^f x_{Mg}^f + w_K^f x_{Kg}^f] \geq Q_g, \quad \forall g \in G \text{ requiring memory} \quad (6)$$

The policy aims to:

$$\min \sum_{f \in F} \sum_{g \in G} (v_{MG}^f x_{Mg}^f + v_K^f x_{Kg}^f) \quad (7)$$

Where  $x_{Mg}^f$  and  $x_{Kg}^f$  are binary variables indicating whether to remove function  $f$ 's model or CUDA kernel from GPU  $g$ . This formulation directly minimizes the total performance value lost due to offloading.

As this optimization problem is also NP-Hard, for efficient scheduling, we use the same greedy algorithm as used for pre-loading that chooses artifacts based on their "value density". This policy executes within microseconds, making it suitable for rapid offloading.

### 4.4 Backbone LLM Sharing

Existing serverful approaches [7, 35, 40] have attempted to share the backbone LLM among multiple LoRA adapters. However, they require both the backbone LLM and LoRA adapters to be maintained within a single process, managed by one CUDA context, and share computational resources for batched inference. Such a design directly conflicts with the serverless paradigm, making it unrealistic for serverless LoRA inference.

Following requirements of serverless computing, each function must run in its own process to maintain security and privacy. Functions can only access data within its own CUDA context and cannot share the backbone with other functions. As each function is allocated and billed for a dedicated amount of CPU and GPU quota, we must restrict sharing strictly to the data layer. All inference computations should execute within their own function instance, using only that instance’s allocated compute resources.

To realize backbone sharing in serverless, we decouple the static backbone tensors from other dynamic components, such as kernels and KV caches. As Fig. 4 shows, the backbone LLM’s tensors are maintained in an individual backbone function. All other components required for inference, including kernels, adapters, and KV caches, are maintained in each function instance’s process (managed by its CUDA context). Firstly, a backbone function instance is created to load the LLM’s tensors into GPUs. Then, these tensors are exposed via CUDA Inter-Process-Communication (IPC) handles, which allow other independent serverless functions to access the same memory region without duplicating backbone tensors.

However, for each LoRA function, simply accessing the raw tensors is insufficient for inference. The Python runtime requires proper mapping between these raw memory regions and the LLM model structure to correctly interpret and utilize the weights within the model’s computational graph during inference. Hence, each LoRA function initializes an empty backbone instance containing the complete model structure but with unpopulated weight tensors. This instance provides the structural mapping needed for the Python runtime to correctly interpret the backbone tensors. Through CUDA IPC, we can assign values to the backbone instance using these tensors with zero-copy.

With this design, though multiple functions access the same backbone tensors, each function executes its own computations independently, utilizing its own GPU resources and maintaining isolated runtime data (including kernel operations, intermediate activations, and KV caches). Thus, *ServerlessLoRA* maintains the security and isolation guarantees demanded by serverless while enabling each function to efficiently execute inference without duplicating the memory-intensive backbone tensors. A GPU can hold hundreds of LoRA functions simultaneously using one backbone LLM.

Furthermore, to achieve LoRA inference in this backbone-adapter decoupling design, *ServerlessLoRA* supports unmerged LoRA inference by keeping the backbone and adapter computations distinct.

Instead of integrating LoRA weights directly into the backbone, which would restrict the use of multiple adapters, *ServerlessLoRA* calculates the attention operation for the backbone and adapter separately and obtains the final attention output by combining the backbone and adapter results.

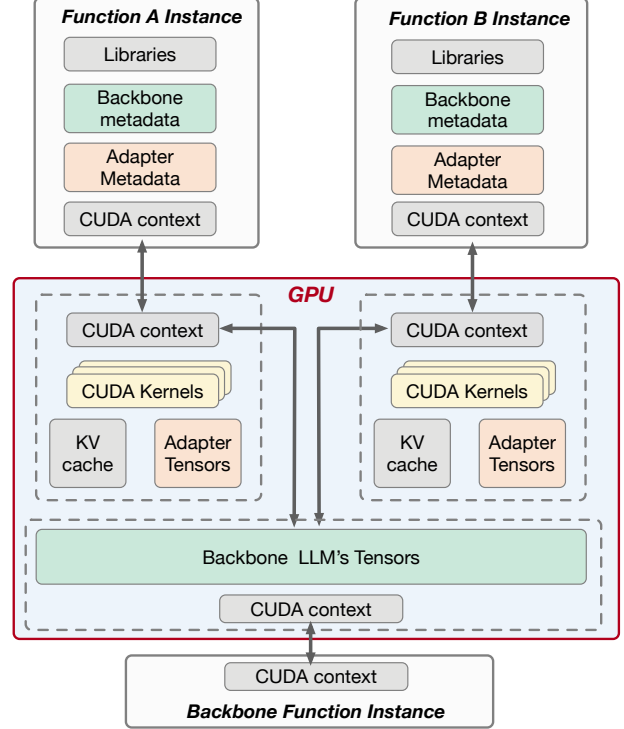


Figure 4. Backbone LLM sharing among function instances.

This division ensures that the shared backbone remains untouched and read-only.

## 5 Implementation

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

**Backbone LLM sharing.** We implement the Backbone sharing based on CUDA IPC handles. After loading the backbone LLM in GPU, we write down the CUDA IPC handle of each layer, which reflects the layer’s memory address. As Python cannot access IPC handles directly, to let inference functions access each layer’s address, we write another CUDA plugin script to get the IPC handles and compile it to a shared object library that can be used for Python. Finally, the IPC handles can be accessed and identified as tensor values. Besides, we modify the model parameter loading function of PyTorch to achieve assigning value to an empty LLM object by letting the accessed tensors point to the LLM with zero copy. As the IPC handles can be continuously and concurrently used for multiple functions, we can share one piece of backbone LLM for multiple functions.

**LoRA inference using the shared backbone LLM.** As the backbone parameters are shared by multiple functions, regular LoRA inference tools like PEFT [26] 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.** As all requests pass through the serverless platform’s controller and all worker nodes’ information are synced to the controller, we deploy the Pre-Loading Scheduler in the controller. The Pre-Loading Agent is deployed in each worker node. We deploy a handler in each container to operate artifacts loading. All communications between these components are through REST API. To accelerate the pre-loading of backbone LLM, we utilize CUDA Streams to load tensors concurrently and CUDA Asynchronous Memory Transfer to overlap loading and GPU transferring.

**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.

## 6 Evaluation

### 6.1 Experimental Setup

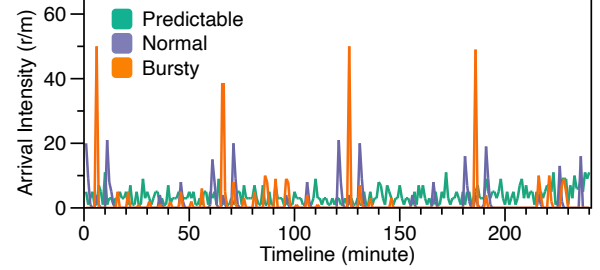
**Testbed.** Our experiments are conducted on two testbeds. The first is a single-node AWS EC2 GPU g6e.48xlarge instance with 384 CPU cores, 1,536 GB memory, and eight NVIDIA L40S GPUs. The second is a multi-node cluster on four AWS g6e.24xlarge instances, with a total of 768 CPU cores, 3,072 GB memory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

**Workload.** To approximate real-world invocation patterns, we utilize production traces from Azure Functions [34], collected over a 14-day period. We categorize Azure Functions traces into three patterns based on the co-variance (CoV) of the request’s inter-arrival time: “Predictable” ( $\text{CoV} \leq 1$ ), “Normal” ( $1 < \text{CoV} \leq 4$ ), and “Bursty” ( $\text{CoV} > 4$ ). Fig. 5 illustrates partial traces of the three patterns. From each pattern, eight 4-hour traces are randomly selected and mapped to individual functions.

**Models and machine learning (ML) Libraries.** We select two backbone LLMs: Llama2-7B and Llama2-13B. Each backbone is augmented with four popular LoRA adapters selected according to download trends on Hugging Face [15]. All inference pipelines are developed using the PyTorch and Transformers.

**Dataset.** We use GSM8K [10], a real-world LLM dataset of human-created problems, as the prompt for each request.

**Baselines.** Two latest serverless and two serverful ML inference serving solutions are selected as baselines: 1) **InstaInfer** [38], a serverless inference system addresses mitigating ML artifacts loading latency for small models by pre-loading. 2) **ServerlessLLM** [16], the state-of-the-art serverless LLM inference framework for minimizing LLM checkpoint loading delay. We also choose two serverful approaches: 3) **dLoRA** [40], which is designed for serving multiple LoRA adapters concurrently. 4) **vLLM** [21], an memory-efficient LLM serving system.



**Figure 5.** Trace example of “Predictable” ( $\text{CoV} \leq 1$ ), “Normal” ( $1 < \text{CoV} \leq 4$ ), and “Bursty” request arrival pattern.

**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 comprehensively evaluate whether the inference is both fast and cheap, we propose the cost-effectiveness metric, defined as  $1/(\text{E2E\_latency} \times \text{Monetary\_Cost})$ .<sup>5</sup> *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 *Serverless-LoRA*.

### 6.2 TTFT and TPOT Evaluation

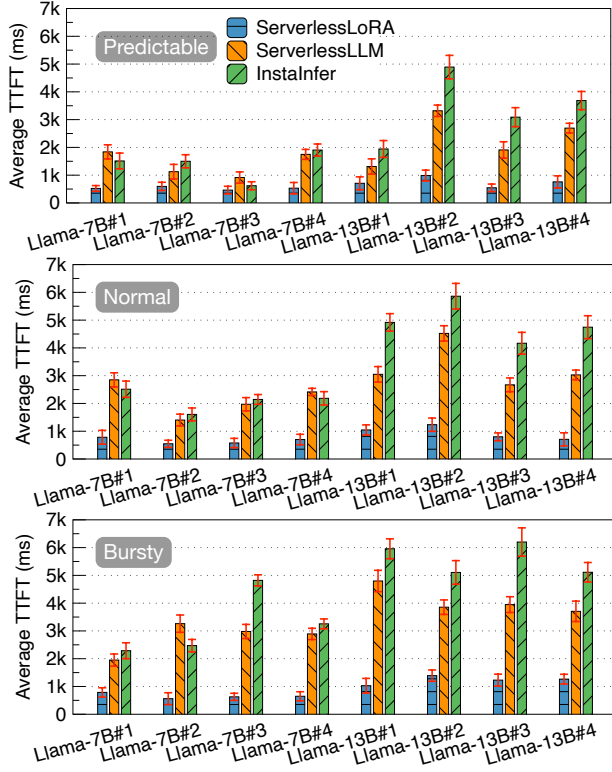
We evaluate the TTFT and TPOT of three serverless solutions on the 16-GPU cluster running four Llama2-7B based LoRA adapter functions and four Llama2-13B based LoRA adapter functions. The evaluation is conducted in three workloads: Predictable, Normal, and Bursty.

**TTFT:** Fig. 6 shows that *ServerlessLoRA* 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 workload, *ServerlessLoRA* 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

<sup>5</sup>Latency and cost alone cannot capture a serverless system’s full value: minimizing latency may lead to GPU over-provisioning, while minimizing cost may incur more cold-starts.

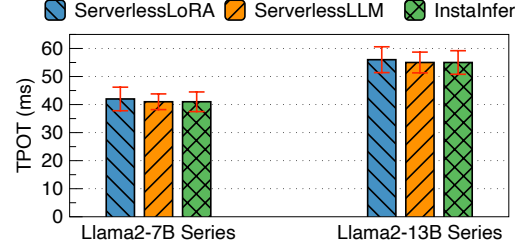




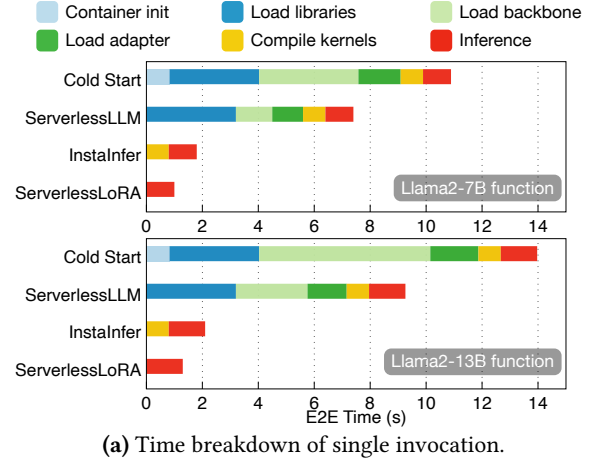
**Figure 6.** Average TTFT of the workloads at “Predictable”, “Normal”, and “Bursty” arrival patterns. Furthermore, all these solutions require loading both the backbone LLM and the LoRA adapter for every function. When a request arrives, they miss the opportunity to use another function’s already-loaded backbone LLM to accelerate artifact loading. Additionally, they ignore the CUDA kernel compilation overhead during 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. 7 shows, *ServerlessLoRA* does not significantly increase TPOT compared with other serverless solutions. Although *ServerlessLoRA*’s TPOT is 12% higher than that of baselines, it still remains within SLO requirements.

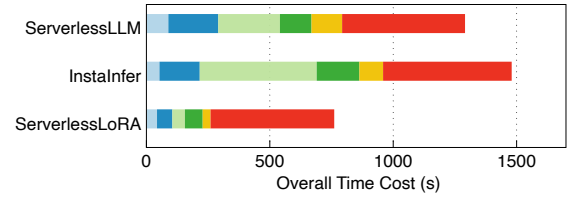
This slightly higher TPOT can be attributed to two main reasons: First, for the goal of minimizing TTFT and improving throughput, *ServerlessLoRA*’s Adaptive Batching Scheduler tends to collect more requests into a batch (while still adhering to SLO constraints). Second, *ServerlessLoRA* reduces replicas of backbone 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. 6.5). As larger batch sizes require more computational resources, *ServerlessLoRA*’s average TPOT is moderately higher than that of InstaInfer and ServerlessLLM, both of which employ fixed and smaller batch sizes.



**Figure 7.** Average TPOT of *ServerlessLoRA* and baselines.



(a) Time breakdown of single invocation.



(b) Time breakdown of the whole workload.

**Figure 8.** Time breakdown analysis.

### 6.3 Time Breakdown Analysis

We evaluate the maximum cold-start mitigation performance of each serverless solution for a single invocation.

We select one Llama2-7B function and one Llama2-13B function. Each function is allocated an entire L40S GPU, ensuring dedicated resource usage. Additionally, functions in each baseline are fully pre-warmed using its respective cold-start mitigation solution. Before experiment, we pre-warm the functions using each solution’s corresponding cold-start mitigation technique, which ensures that each solution is operating under the most favorable conditions for reducing cold-start latency. Finally, we invoke the functions and record their cold-start latencies to evaluate the best-case cold-start performance that each solution can deliver.

Fig. 8a shows the time breakdown of a single cold-start invocation for each function. Only *ServerlessLoRA* can fully eliminate all cold-starts. In the best-case scenario, since all LLM artifacts are pre-loaded, functions can begin inference immediately—making cold-starts as fast as warm-starts. For InstaInfer, as it misses the opportunity of pre-loading CUDA kernels, even if all libraries and models are pre-loaded, the

request has 9% cold-start latency remaining. For ServerlessLLM, it only addresses the loading of LLM checkpoints, while leaving the substantial cold-start latency unaddressed.

As single invocation’s cold-start reduction cannot thoroughly reflect the acceleration performance for the whole workload, we further evaluate the cumulative time cost and breakdown of each solution running the “Normal” workload in Fig. 8b. *ServerlessLoRA* significantly reduces latency of loading each artifact, especially the backbone LLM. While InstaInfer and ServerlessLLM introduces higher cumulative cold-start latency than inference, indicating their limitations in serving LoRA LLMs.

#### 6.4 Cost-Effectiveness

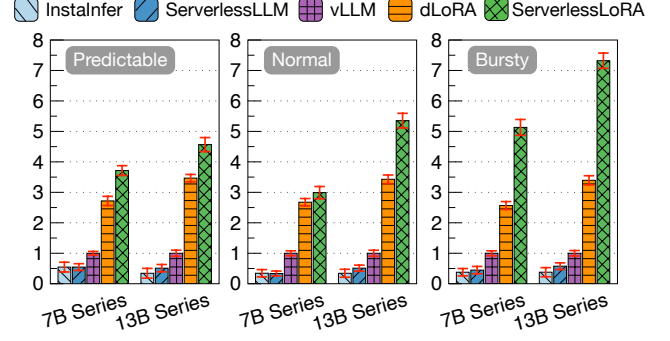
To evaluate *ServerlessLoRA*’s cost-effectiveness on achieving faster and cheaper LLM inference, we need to jointly consider the effect of E2E latency and monetary cost. Therefore, we show the cost-effectiveness of both serverless and serverful solutions. For clarity, we set vLLM as the baseline and present the relative cost-effectiveness compared with vLLM. We use Alibaba Cloud serverless pricing rule [9] for the monetary cost.

Fig. 9 shows that *ServerlessLoRA* outperforms both serverless and serverful inference baselines with each type of workload. Compared with vLLM and dLoRA, which serve requests in long-running VMs and thereby have zero cold-starts, *ServerlessLoRA* significantly reduces the monetary cost by several times while only adding little cold-start latency. Compared with serverless solutions, due to the pre-loading and backbone sharing, *ServerlessLoRA* reduces both cold-starts latency and monetary cost, thereby outperforms InstaInfer and ServerlessLLM up to 12.7 $\times$  and 19.3 $\times$ .

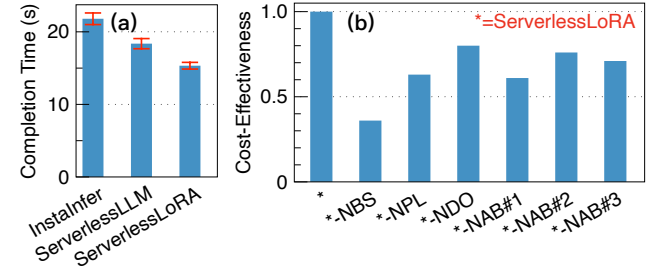
Meanwhile, for serverless baselines, they perform worse with Llama2-13B compared to Llama2-7B. The reason for this phenomenon is that larger models demand longer loading times and more GPU resources, which increases E2E latency and monetary costs, making them less cost-efficient. InstaInfer, designed for models with millions of parameters (e.g., ResNet), exhibits the highest TTFT and monetary cost when applied to LLMs with billions of parameters. This finding underscores our motivation: existing serverless solutions cannot be directly applied for LLM serving.

#### 6.5 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, *ServerlessLoRA* outperforms both ServerlessLLM and InstaInfer, improving the maximum output token throughput 1.65 $\times$ , the maximum batch size 2.28 $\times$ , and the completed requests throughput up to 3.02 $\times$ .



**Figure 9.** Cost-effectiveness of *ServerlessLoRA* and baselines running the Predictable, Normal, and Bursty workloads.



**Figure 10.** Throughput and ablation study evaluation.

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

Furthermore, *ServerlessLoRA*’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 *ServerlessLoRA*’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. 10 (a) shows that *ServerlessLoRA* achieves the shortest completion time. Consequently, even at its peak batch size, when resource contention is highest, *ServerlessLoRA* sustains superior inference speed.

#### 6.6 Ablation Study

To evaluate the effectiveness of *ServerlessLoRA*’s each component (Backbone Sharing, Pre-Loading, Dynamic Offloading, and Adaptive Batching) separately, we conduct an ablation study on the 4-node GPU cluster.

We compare *ServerlessLoRA* with its four variants:

***ServerlessLoRA*-NBS:** *ServerlessLoRA* without the Backbone Sharing mechanism. In this variant, each function must independently hold a complete backbone LLM.

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

Workload	E2E Latency (ms)			Cost (\$)			Cost-Effectiveness (relative)		
	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)
<i>ServerlessLoRA</i>	<b>2922 (3166)</b>	<b>3061 (3338)</b>	<b>3050 (3631)</b>	<b>4.66 (7.30)</b>	<b>5.54 (5.87)</b>	<b>3.35 (4.16)</b>	<b>3.71 (4.57)</b>	<b>2.99 (5.35)</b>	<b>5.13 (7.32)</b>

**Table 2.** Peak throughput of each serverless solution

	Throughput (Token/s)	Peak Batch Size	Throughput (Request/s)
<i>ServerlessLoRA</i>	547	73	4.76
ServerlessLLM	331	32	1.72
InstaInfer	331	32	1.48

**ServerlessLoRA-NPL:** *ServerlessLoRA* without the Pre-Loading Scheduler. In this variant, none LLM artifacts are pre-loaded.

**ServerlessLoRA-NDO:** *ServerlessLoRA* without Dynamic Offloading. In this variant, when 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.

**ServerlessLoRA-NAB:** *ServerlessLoRA* without the Adaptive Batching Scheduler. In this variant, we set each function’s batch size and batch delay 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 *ServerlessLoRA*-NAB #1-#3.

We run a 4-hour “Normal” workload of 4 Llama2-7B series functions and 4 Llama2-13B series functions. Fig. 10 (b) shows, *ServerlessLoRA* achieves the highest cost-effectiveness, compared with each variant. Among all variants, *ServerlessLoRA*-NBS performs worst, indicating that the backbone sharing mechanism plays the most crucial role in reducing TTFT and monetary cost.

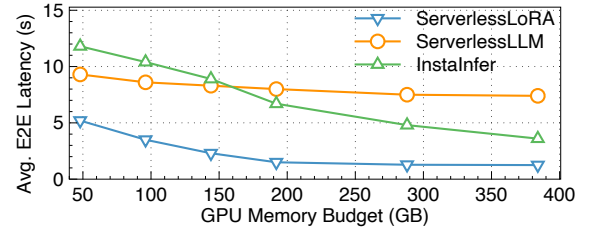
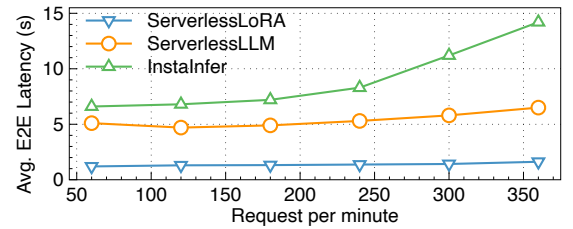
We show the detailed results of each variants’ TTFT, E2E latency, and monetary cost in Table. 3. As the result shows, *ServerlessLoRA* achieves the least TTFT, E2E latency, and monetary cost. It’s worth mentioning that although *ServerlessLoRA*-NAB #2 and #3 achieve similar TTFT as *ServerlessLoRA*, the fixed batching causes significant resource contention among requests for different functions, thereby slowing down the inference speed and increasing E2E latency and monetary cost.

## 6.7 Scalability

We evaluate *ServerlessLoRA*’s scalability with two common metrics: 1) **Strong scalability**, which increases GPU resources while keeping the total workload fixed. 2) **Weak**

**Table 3.** Ablation study of *ServerlessLoRA*.

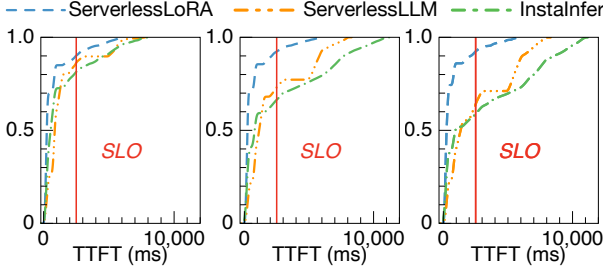
<i>ServerlessLoRA</i> Variants	TTFT (ms)	E2E Latency (ms)	Monetary Cost (\$)
<i>ServerlessLoRA</i>	576	2977	7.53
<i>ServerlessLoRA</i> -NBS	2608	4959	12.55
<i>ServerlessLoRA</i> -NPL	1345	3747	9.48
<i>ServerlessLoRA</i> -NDO	1047	3328	8.42
<i>ServerlessLoRA</i> -NAB #1	1386	3799	9.61
<i>ServerlessLoRA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

**(a)** Strong scalability evaluation.**(b)** Weak scalability evaluation.**Figure 11.** Scalability comparison of *ServerlessLoRA* and other serverless solutions.

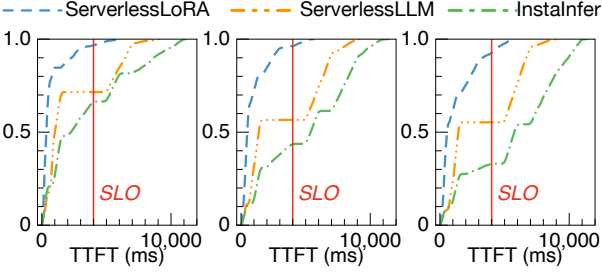
**scalability**, which proportionally increases both workloads and GPU resources.

Fig. 11a shows, for the workload that contains all 8 Llama functions, with increasing GPU memory, *ServerlessLoRA* consistently outperforms other serverless solutions, indicating its efficient GPU utilization whether GPU resources are limited or abundant. When more GPU memory is available, *ServerlessLoRA* effectively converts these resources into faster inference.

Fig. 11b shows, in *ServerlessLoRA*, the average E2E latency of each request is stable when the GPU resource and workload increases proportionally, while InstaInfer’s cost-effectiveness significantly decreases under heavy workload.



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



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

**Figure 12.** TTFT CDF of *ServerlessLoRA* and baselines running the Predictable, Normal, and Bursty workloads.

This shows that *ServerlessLoRA* can scale effectively for large deployments, handling increased workloads without slowing down or running into resource conflicts.

### 6.8 SLO Violation

As *ServerlessLoRA* aims to minimize TTFT and monetary cost without violating SLO, we evaluate the SLO violation rate running the Predictable, Normal, and Bursty workload. We set the TTFT SLO standard following the same setting of ParaServe [25]: 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. 12 shows that *ServerlessLoRA* 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 indicates that *ServerlessLoRA*’s outperformance over current serverless solutions does not increase SLO violation rate.

### 6.9 Overhead

We measured the resource and latency overhead of *ServerlessLoRA* when running the above workloads. The Pre-Loading Scheduler and the Adaptive Batching Scheduler introduce 1ms additional latency respectively. The overall scheduling overhead of *ServerlessLoRA* is less than 6ms under the heaviest workload. The backbone sharing does not introduce any latency overhead, which means that *ServerlessLoRA* can

share the backbone among functions without slowing down inference speed.

As for resource consumption, all scheduling components costs 1 CPU core and 300MB host memory in total. The backbone sharing introduces 473 MB GPU memory overhead. This overhead arises because each process (backbone and adapter) must maintain its own CUDA context, leading to duplicated GPU resource usage. Compared with the saved GPU memory (14GB–80GB), this overhead is negligible.

## 7 Related Work

**LLM serving:** Recent studies focus on accelerating LLM inference and increasing throughput. Orca [42] batches requests at iteration level to minimize queuing time, while FlashAttention [12] reduces IO complexity. SpecInfer [28] utilizes speculative decoding to reduce latency. DeepSpeed [6] and AlpaServe [24] leverage multi-GPU parallelization. For improved throughput, vLLM [21], InfiniGen [22], and MoonCake [31] optimize KV cache management, while FlexGen [36] and LLM-in-a-flash [4] off-load data to host memory. DistServe [49] and SARATHI [1] disaggregate pre-filling and decoding to maximize GPU utilization. *ServerlessLoRA* is designed to be orthogonal to these LLM serving solutions, as they address optimizing the inference stage while *ServerlessLoRA* focuses on mitigating the cold-starts before inference.

**Multi-LoRA LLM inference:** Unlike general LLMs, multi-LoRA LLMs introduce new challenges in efficiently sharing the backbone model without incurring additional memory overhead or degrading inference speed. Punica [7] and S-LoRA [35] support batching requests of different adapters on the same backbone LLM. Besides multi-LoRA support, dLoRA [40] also considers the inference acceleration of merging the LoRA adapter into the backbone LLM. However, these serverful solutions run in long-running VMs and requires running models in a single process, violating the isolation requirement of serverless computing. Instead, *ServerlessLoRA* runs each LoRA adapter in an independent function instance. The adapter model, KV cache, and other data are isolated between functions.

**Optimizing serverless inference:** serverless ML inference has been well studied [3, 8, 13, 18–20, 23]. However, these studies ignore the significant delay caused by model loading. Some approaches group requests into batches to improve throughput [2, 41, 46]. However, as they are designed for common serverless applications, they cannot flexibly adjust the batch size and delay to avoid slowing down inference. *ServerlessLLM* [16], Medusa [45], ParaServe [25], and λScale [44] address the cold start problem in serverless inference optimized for LLM. They are orthogonal and compatible with *ServerlessLoRA*. AsyFunc [29], Tetris [23], and Optimus [18] accelerates the model-loading latency, while InstaInfer [38] fully mitigates the latency of loading all ML artifacts. *ServerlessLLM* [16], Medusa [45], ParaServe [25],



and  $\lambda$ Scale [44] accelerate the LLM checkpoint loading and compiling. However, all these solutions require maintaining a complete LLM within each function instance, causing extremely high resource and monetary cost and, meanwhile, limiting the scalability.

## 8 Conclusion

This paper presented *ServerlessLoRA*, 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. *ServerlessLoRA* propose secure backbone LLM sharing, comprehensive LoRA artifact pre-loading, contention-aware adaptive batching, and dynamic GPU memory offloading. Extensive evaluation demonstrates that *ServerlessLoRA* 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 at scale.

## References

- [1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. *arXiv preprint arXiv:2308.16369* (2023).
- [2] Ahsan Ali, Riccardo Pincirolì, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine Learning Inference Serving on Serverless Platforms With Adaptive Batching. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–15.
- [3] Ahsan Ali, Riccardo Pincirolì, Feng Yan, and Evgenia Smirni. 2022. Optimizing Inference Serving on Serverless Platforms. *Proc. the VLDB Endowment* 15, 10 (2022).
- [4] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. In *Proc. the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*. 12562–12584.
- [5] Inc. Amazon Web Services. 2025. Amazon Bedrock - Build Generative AI Applications with Foundation Models. <https://aws.amazon.com/bedrock/> Accessed: 2025-05-15.
- [6] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models At Unprecedented Scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–15.
- [7] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2024. Punica: Multi-Tenant LoRA Serving. *Proc. Machine Learning and Systems (MLSys)* 6 (2024), 1–13.
- [8] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. 2022. Sla-driven ML Inference Framework for Clouds with Heterogeneous Accelerators. *Proc. Machine Learning and Systems (MLSys)* 4 (2022), 20–32.
- [9] Alibaba Cloud. 2025. Billable items and billing methods - Function Compute - Alibaba Cloud. <https://www.alibabacloud.com/help/en/functioncompute/fc-2-0/product-overview/billing-overview> Last Updated: Jan 20, 2025. Accessed: May 16, 2025.
- [10] 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. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168* (2021).
- [11] NVIDIA Corporation. 2025. DGX Platform: Built for Enterprise AI | NVIDIA. <https://www.nvidia.com/en-us/data-center/dgx-platform/> Accessed: 2025-05-15.
- [12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *Advances in Neural Information Processing Systems (NeurIPS)* 35 (2022), 16344–16359.
- [13] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a Diet. In *Proc. the 11th ACM Symposium on Cloud Computing (SoCC)*. 45–59.
- [14] Jonatan Enes, Roberto R Expósito, and Juan Touriño. 2020. Real-time resource scaling platform for big data workloads on serverless environments. *Future Generation Computer Systems* 105 (2020), 361–379.
- [15] Hugging Face. 2025. Hugging Face: PEFT LLaMA2 Models (Sort By Most Downloads). <https://huggingface.co/models?library=peft&sort=downloads&search=llama2> Accessed: 2025-05-08.
- [16] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Locality-Enhanced Serverless Inference for Large Language Models. *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2024).
- [17] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T Kandemir, and Chita R Das. 2020. Fifer: Tackling Underutilization in the Serverless Era. In *The 21st International Middleware Conference (Middleware)*.
- [18] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. 2024. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In *Proc. the 19th European Conference on Computer Systems (EuroSys)*. 1039–1053.
- [19] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2021. Amps-inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency. In *Proc. the 50th International Conference on Parallel Processing (ICPP)*. 1–12.
- [20] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proc. the 2021 International Conference on Management of Data (SIGMOD)*. 857–871.
- [21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. the 29th Symposium on Operating Systems Principles (SOSP)*. 611–626.
- [22] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient Generative Inference of Large Language Models with Dynamic {KV} Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 155–172.
- [23] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-Efficient Serverless Inference Through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC)*.
- [24] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 663–679.
- [25] Chiheng Lou, Sheng Qi, Chao Jin, Dapeng Nie, Haoran Yang, Xuanzhe Liu, and Xin Jin. 2025. Towards Swift Serverless LLM Cold Starts with ParaServe. *arXiv preprint arXiv:2502.15524* (2025).

- [26] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: Parameter-Efficient Fine-Tuning methods. <https://github.com/huggingface/peft>.
- [27] Tuna Han Salih Meral, Enis Simsar, Federico Tombari, and Pinar Yarnadag. 2024. Clora: A contrastive approach to compose multiple lora models. *arXiv preprint arXiv:2403.19776* (2024).
- [28] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. 2024. Specinfer: Accelerating Large Language Model Serving with Tree-Based Speculative Inference and Verification. In *Proc. The 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 932–949.
- [29] Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. 2023. AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions. In *Proc. the ACM Symposium on Cloud Computing (SoCC)*. 324–340.
- [30] David Pisinger and Paolo Toth. 1998. Knapsack problems. *Handbook of Combinatorial Optimization* (1998), 299–428.
- [31] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. 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)*. 155–170.
- [32] Qwen. 2025. Qwen Chat. <https://chat.qwen.ai/>. Accessed: 2025-05-15.
- [33] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated Model-less Inference Serving. In *Proc. 2021 USENIX Annual Technical Conference (USENIX ATC)*. 397–411.
- [34] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. 2020 USENIX Annual Technical Conference (USENIX ATC)*. 205–218.
- [35] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. 2024. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. *Proc. Machine Learning and Systems (MLSys)* (2024).
- [36] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *International Conference on Machine Learning (ICML)*. PMLR, 31094–31116.
- [37] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1348–1362.
- [38] Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. 2024. Pre-Warming is Not Enough: Accelerating Serverless Inference with Opportunistic Pre-Loading. In *Proc. the 2024 ACM Symposium on Cloud Computing (SoCC)*. 178–195.
- [39] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
- [40] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. {dLoRA}: Dynamically Orchestrating Requests and Adapters for {LoRA}{LLM} Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 911–927.
- [41] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. 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)*. 768–781.
- [42] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for {Transformer-Based} Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 521–538.
- [43] Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2023. Libra: Harvesting idle resources safely and timely in serverless clusters. In *Proc. the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 181–194.
- [44] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Yue Cheng, Wei Wang, Ao Wang, and Ruichuan Chen. 2025. λScale: Enabling Fast Scaling for Serverless Large Language Model Inference. *arXiv preprint arXiv:2502.09922* (2025).
- [45] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. 2025. Medusa: Accelerating Serverless LLM Inference with Materialization. In *Proc. the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 653–668.
- [46] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. {MARK}: Exploiting Cloud Services for {Cost-Effective},{SLO-Aware} Machine Learning Inference Serving. In *Proc. 2019 USENIX Annual Technical Conference (USENIX ATC)*. 1049–1062.
- [47] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*.
- [48] Ming Zhong, Yelong Shen, Shuohang Wang, Yadong Lu, Yizhu Jiao, Siru Ouyang, Donghan Yu, Jiawei Han, and Weizhu Chen. 2024. Multi-lora composition for image generation. *arXiv preprint arXiv:2402.16843* (2024).
- [49] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 193–210.
- [50] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. 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)*. 1–14.