CELLServe: An SLO-Aware and Cost Efficient LLMs Serving System for Serverless Computing Environments

ZEJIAN WANG*, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
NAN LIN*, ENSTA Paris, Palaiseau, France and SJTU Paris Elite Institute of Technology, Shanghai Jiao Tong University, Shanghai, China
ZINUO CAI, Shanghai Jiao Tong University, Shanghai, China
RONGBO MA, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
RUHUI MA†, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
HAIBING GUAN, School of Software, Shanghai Jiao Tong University, Shanghai, China
RAJKUMAR BUYYA, Cloud Computing and Distributed Systems Laboratory, The University of Melbourne, Melbourne, Australia

Large Language Models (LLMs) have enabled diverse AI applications. However, LLMs inference impose unprecedented computational and memory overhead, creating an inherent trade-off between latency Service Level Objectives (SLOs) and resource constraints. Serverless computing, with on-demand provisioning and pay-as-you-go billing, is becoming a promising paradigm for LLM serving. But existing solutions fail to integrate state-of-the-art inference optimizations, resulting in suboptimal GPU utilization and prolonged latency. While Prefill-Decode (PD) disaggregation combined with continuous batching has resolved such inefficiencies in traditional cloud deployments, migrating these techniques to serverless makes two challenges particularly pronounced: (1) SLO-constrained resource provisioning for independently scaling *prefill* and *decode phase* functions, and (2) function lifespan management to mitigate resource waste from continuous batching-induced prolonged instance lifespans. To tackle these issues, we propose *CELLServe*, an SLO-aware and cost-efficient serverless LLM serving system that pioneers integrating PD disaggregation and continuous batching into serverless platforms. *CELLServe* formalizes SLO-constrained joint resource provisioning as an optimization problem with a dedicated algorithm, and introduces an opportunistic instance merging strategy for *decode phase* functions to reclaim fragmented resources. Comprehensive evaluations on five mainstream LLMs and real-world traces show that *CELLServe* achieves 1.85×–1.92×higher request throughput than baselines under identical SLOs and GPU budgets, while sustaining high resource efficiency under dynamic workloads.

*Both authors contributed equally to this research.
†Corresponding author.

Authors' Contact Information: Zejian Wang, School of Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: zejianwang@sjtu.edu.cn; Nan Lin, ENSTA Paris, Palaiseau, Île-de-France, France and SJTU Paris Elite Institute of Technology, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: nan.lin@ensta-paris.fr; Zinuo Cai, Shanghai Jiao Tong University, Shanghai, China; e-mail: kingczn1314@sjtu.edu.cn; Rongbo Ma, School of Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: marongbo@sjtu.edu.cn; Ruhui Ma, School of Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: ruhuima@sjtu.edu.cn; Haibing Guan, School of Software, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: hbguan@sjtu.edu.cn; Rajkumar Buyya, Cloud Computing and Distributed Systems Laboratory, The University of Melbourne, Melbourne, Victoria, Australia; e-mail: rbuyya@unimelb.edu.au.

ACM 1544-3973/2026/7-ART
https://doi.org/10.1145/3830086

1 Introduction

Recent advancements in Large Language Models (LLMs) have enabled AI applications across various domains [6, 28, 38]. However, LLM inference is inherently resource-intensive as mainstream models have billions of parameters, thus imposing an unprecedented surge in computational and memory resource consumption [4]. Different users have different Service Level Objectives (SLOs), as some prioritize real-time responsiveness, while others can tolerate higher latency in exchange for improved accuracy [2, 29]. To meet these diverse demands, service providers are increasingly exploring customized LLM solutions tailored to their domain-specific needs, optimizing for factors such as latency-optimized inference and cost-efficient resource utilization [21, 47].

Serverless computing has emerged as a promising solution for LLM service providers, with its core attributes of on-demand resource provisioning and pay-as-you-go billing [8, 15, 20, 25], delivering significant improvements in resource efficiency for LLM serving. Unlike traditional cloud infrastructure, which suffers from chronic over-provisioning of resources to accommodate peak traffic and severe under-utilization during low-load periods, serverless architectures encapsulate each LLM inference instance as a "function" that scales elastically with fluctuating request volumes [8, 23, 39]. This on-demand and pay-as-you-go allocation scheme ensures that resources are consumed only during active request processing. Therefore, serverless has become an ideal paradigm for LLM inference.

However, existing serverless LLM serving works primarily focus on mitigating the cold start problem in serverless computing, while overlooking the integration of state-of-the-art LLM inference optimizations into the serverless platform [5, 8, 15, 36]. Specifically, these solutions are anchored in a paradigm that encapsulates the entire LLM inference process within a single serverless function while adopting a static batching strategy to enhance system throughput. This paradigm fails to account for the inherent heterogeneity of the LLM inference process, resulting in suboptimal serving performance that manifests in two key limitations. First, it leads to inefficient GPU resource utilization due to the overlooked asymmetry between the compute-intensive *prefill* and memory-intensive *decode* stages, a mismatch that cannot be resolved by a one-size-fits-all function encapsulation scheme [48]. Second, it causes prolonged end-to-end latency due to static batching's rigid synchronization mechanism where all requests in a batch must wait for the longest-running autoregressive decoding task to complete [42]. These shortcomings have been widely discussed in prior research. To address these limitations, *Prefill-Decode* (PD) disaggregation integrated with continuous batching has been proven to be an effective LLM inference optimization approach in traditional cloud-based LLM deployments, a combination that effectively eliminates the aforementioned performance bottlenecks [3, 16, 30].

Nonetheless, migrating PD disaggregation and continuous batching to the serverless paradigm is non-trivial. Compared with cloud-native or container deployments that maintain long-lived provisioned replicas, the elastic traits of serverless make phase-specific provisioning and instance lifecycle management joint online optimization problems. In fact, the adaptation and integration of these two techniques into serverless platforms spawn two serverless-amplified challenges that become particularly pronounced under serverless execution. C1: ***Function Resource Provisioning with SLO-Constraints***. During serverless function scaling, resource allocation becomes particularly challenging in PD disaggregation serving setups, where *prefill* and *decode phase* functions scale independently and exhibit divergent resource demands that make it difficult to reconcile predefined SLOs with resource constraints. Our experiments in Section 2.2 demonstrate that under identical SLO constraints and fixed total resources, different resource provisioning strategies yield markedly different levels of system throughput. C2: ***Function Lifespan Management for Resource Efficiency***. During *decode phase*, continuous batching enables the system to admit new requests and evict completed ones without waiting for the entire batch to finish decoding. However, when request load declines, long-sequence autoregressive decoding forces instances to remain active for extended lifespans, yet the resources occupied by these instances become underutilized due

to reduced batch sizes. Our experiments in Section 2.2 further corroborate this resource utilization inefficiency during workload transitions.

To address the aforementioned challenges, we propose *CELLServe*, a SLO-aware and Cost Efficient serverless LLM serving system. Our work pioneers a serverless-tailored LLM inference serving solution by effectively integrating PD disaggregation and continuous batching into the serverless platform, which aligns with the serverless dynamic scaling characteristics. First, we formalize the SLO-constrained joint resource provisioning for scaling *prefill* and *decode* functions as a mathematical optimization problem, and design an efficient resource provisioning algorithm. This algorithm leverages the workload's Requests Per Second (RPS) rate and offline model profiling data to dynamically allocate GPU compute and memory resources to the two scaling functions, balancing resource utilization with strict compliance to latency SLOs and resolving the core inefficiencies highlighted in Challenge 1. Second, we introduce an opportunistic instance merging strategy that merges low-utilization *decode* instances to reclaim fragmented resources and timely release unneeded ones. This strategy directly addresses the resource waste dilemma outlined in Challenge 2.

We comprehensively evaluate *CELLServe* on five representative LLMs (LLaMA2-7B, Mistral-7B, GLM4-9B, Qwen3-14B, and Qwen3-0.6B) and real-world inference workloads from the Azure LLM Inference Trace. The results show that *CELLServe* consistently delivers higher throughput and lower latency than existing serverless platforms across both conversation and code-generation scenarios. Under the same SLO requirements and fixed GPU budgets, *CELLServe* achieves up to 1.85×, 1.71×, and 1.78× higher request throughput than OpenWhisk, OpenFaaS, and ServerlessLLM, respectively, as well as up to 4.3× per-device throughput improvement over static allocation baselines, while maintaining strict Time to First Token (TTFT) and Time per Output Token (TPOT) guarantees. Moreover, its adaptive resource provisioning and opportunistic instance merging sustain high streaming multiprocessor (SM) utilization and memory efficiency even under dynamic and bursty load patterns.

Our contributions can be summarized as follows:

- We systematically identify two challenges amplified by serverless execution that emerge when adapting PD disaggregation and continuous batching to serverless LLM serving platforms under the occupancy-based billing model: SLO-constrained joint provisioning for independently scaled *prefill phase*/*decode phase* functions, and timely lifecycle management of underutilized *decode phase* functions introduced by continuous batching.
- We formally model the SLO-constrained joint resource provisioning problem for independently scaled *prefill* and *decode* phase functions as a mathematical optimization problem, and propose an efficient resource provisioning algorithm that leverages workload RPS metrics and offline model profiling data to balance GPU resource utilization with strict latency SLO compliance.
- We design an opportunistic instance merging strategy tailored for *decode phase* functions, which reclaims fragmented GPU resources from low-utilization instances and eliminates idle resource overhead, thereby mitigating the resource waste dilemma induced by continuous batching and fluctuating workloads.
- We conduct comprehensive evaluations of *CELLServe* across five representative LLMs and realistic Azure LLM inference traces, showing that *CELLServe* achieves up to 1.85×, 1.71×, and 1.78× higher throughput than OpenWhisk, OpenFaaS, and ServerlessLLM, respectively, together with up to 60.7% lower end-to-end latency while maintaining high GPU memory and SM utilization through adaptive provisioning and instance merging.

2 Background and Motivation

2.1 LLMs Serving

LLM Inference. Modern LLMs [1, 24, 45] are built on the transformer architecture, which relies on attention and Feed-Forward Neural (FFN) network to interpret input prompts and generate tokens. LLM inference consists

of two sequential phases: the *prefill phase* and the *decode phase*. During the *prefill phase*, the LLM processes all input prompt tokens in parallel with intensive matrix-matrix multiplications, initializes the Key-Value Cache (KV Cache) to store intermediate attention results, and generates the first output token [18]. After the *prefill phase* completes, the *decode phase* leverages the precomputed KV Cache to produce one token per forward iteration, consuming substantial memory and IO bandwidth to maintain the growing KV Cache. Therefore, the *prefill phase* is compute-bound and quantified by the Time to First Token (TTFT) metric that reflects the latency of the first output token generation; conversely, the *decode phase* is memory-bound, with its performance measured by Time per Output Token (TPOT), which denotes the latency between consecutive token generations.

PD Disaggregation Serving. The heterogeneous resource demands of *prefill phase* and *decode phase* create inherent bottlenecks in monolithic LLM serving architectures, leading to both resource underutilization and suboptimal latency. To address this mismatch, PD disaggregation serving decouples the two inference phases into different clusters, where *prefill phase* is deployed on high-throughput GPU clusters to minimize TTFT and the *decode phase* resides on memory-optimized nodes equipped with high-bandwidth storage and cache-coherent interconnects. Representative work in this field includes [48] and Splitwise[27]. Furthermore, another strategy to address this resource mismatch is to leverage GPU spatial multiplexing technology, like MPS[1], MIG[2] to provision resources for the two inference stages within a single GPU cluster, which is contemporaneous with our work. Representative implementations including DuetServe [9] and SpaceServe [46]. Overall, PD disaggregation serving has been widely validated in traditional cloud infrastructure.

Batching. Traditional static batching improves GPU utilization by grouping multiple inference requests into a single batch for inference [32, 41]. However, this design neglects autoregressive nature of LLM inference and incurs unnecessary latency overheads: all batch requests must synchronize token generation iterations and wait for the longest-running request to finish. Current serving systems utilize a batching technique known as continuous batching, first introduced by Orca [42]. In contrast to static batching's synchronous token generation, continuous batching enables the system to admit new requests into an ongoing batch and evict requests that have generated termination tokens, thereby eliminating end-to-end latency penalties associated with waiting for the batch's longest-running request and boosting GPU utilization.

2.2 Challenges

Integrating PD disaggregation with continuous batching into serverless LLM serving is non-trivial, despite serverless's inherent elasticity and cost efficiency for LLM workloads and the two techniques' proven effectiveness in traditional cloud LLM deployments. In fact, this cross-paradigm integration makes two distinct challenges particularly pronounced in serverless environments, as the intrinsic traits of serverless constrain the effectiveness of these LLM serving optimizations.

The first challenge lies in ***Function Resource Provisioning with SLO-Constraints***. Serverless's inherent occupancy-billing traits demand achieving optimal resource utilization while adhering to SLOs. Nonetheless, the PD disaggregation serverless serving complicates this target, where *prefill phase* and *decode phase* functions need to scale independently to match their divergent resource demands. To quantify this complexity and validate the performance impact of mismatched resource provisioning, we conduct a controlled experiment on an NVIDIA GeForce RTX 4090 (48GiB VRAM) to evaluate how distinct resource allocation strategies affect end-to-end system performance.

Specifically, we simulate an inference workflow under a fixed two-GPU setting with prompt sequence length = 256 tokens and *decode phase* generation length = 32 tokens, with latency SLOs set as TTFT $\leq$ 500 ms and TPOT $\leq$ 25 ms. We test a range of configurations with different allocations of active Streaming Multiprocessor

[1] https://docs.nvidia.com/deploy/mps/index.html
[2] https://www.nvidia.com/en-us/technologies/multi-instance-gpu/

Table 1. SLO-Compliant Single-Instance Profiles for Provisioning

Phase	Batch Size	SMs Utilization (%)	Time (ms)	Memory (GiB)
Prefill	64	100	443.92	23.95
Prefill	32	75	293.74	18.31
Prefill	32	50	390.37	18.31
Decode	128	100	24.62	41.28
Decode	64	50	24.73	28.42
Decode	64	25	24.71	28.42

Note: SMs utilization is the percentage of active SM threads; Time is TTFT for *prefill phase* and TPOT for *decode phase*.

(SM) threads and batch sizes, then filter results to retain only those that satisfy the predefined SLOs, as shown in Table. 1. From these profiles, we construct three representative provisioning schemes under two GPUs scenario: (1) a phase-dedicated baseline, in which a single *prefill phase* instance and a single *decode phase* instance each exclusively occupy one GPU; (2) an intermediate comparison set that increases only the number of *prefill phase* replicas while keeping the *decode phase* side unchanged; and (3) a co-located packing scheme that exploits the complementary resource demands of *prefill phase* and *decode phase*.

- **Scheme (1):** A single *prefill phase* function (batch size = 64, SM utilization = 100%) runs on GPU 0, while a single *decode phase* function (batch size = 128, SM utilization = 100%) runs on GPU 1.
- **Scheme (2):** Two *prefill phase* functions run on GPU 0, each with batch size = 32 and SM utilization = 50%, while a single *decode phase* function (batch size = 128, SM utilization = 100%) runs on GPU 1.
- **Scheme (3):** On each of GPU 0 and GPU 1, we co-deploy one *prefill phase* function (batch size = 32, SM utilization = 75%) and one *decode phase* function (batch size = 64, SM utilization = 25%).

Across the three provisioning schemes, we report the aggregate *prefill phase* throughput, aggregate *decode phase* throughput, and total GPU memory footprint of the two-GPU system as follows: Scheme (1) achieves 144.17 RPS for *prefill phase* and 5203.25 Token Per Second (TPS) for *decode phase*, with a combined memory footprint of 65.23 GiB; Scheme (2) delivers 163.95 RPS for *prefill phase* and 5182.18 TPS for *decode phase*, with a combined memory footprint of 77.90 GiB; Scheme (3) yields 217.88 RPS for *prefill phase* and 5140.56 TPS for *decode phase*, with a combined memory footprint of 93.46 GiB. With Scheme (1) established as the baseline (full SM utilization and maximal batch sizes for both phases), Scheme (2) improves *prefill phase* throughput by 13.72% relative to the baseline. Scheme (3) further improves *prefill phase* throughput by 51.13% by allocating 75% SM to each *prefill phase* instance and pairing it with a low-SM, memory-oriented *decode phase* profile on the same GPU. Notably, decode TPS changes by less than 0.5% across Schemes (2) and (3), showing that this co-located packing strategy raises prefill capacity without materially affecting decode throughput.

Challenge I

For serverless PD-disaggregated LLM serving, determining the optimal compute and memory resource provisioning strategy when scaling functions to meet SLO and resource constraints remains non-trivial, as varying strategies have been shown to exert a substantial impact on overall system performance.

The second challenge lies in ***Function Lifespan Management for Resource Efficiency***. This challenge stems from the inherent conflict between continuous batching's scheduling logic during *decode phase* and serverless's demand for timely resource release to maintain cost-efficiency [12]. In high-load scenarios, continuous batching excels at maximizing instance utilization by dynamically admitting new requests and evicting completed ones during autoregressive decoding, eliminating the idle wait overhead of static batching. However, the mechanism

breaks down during the transition from high loads to low loads, where long-sequence decode requests are becoming the dominant workload. These long-sequence autoregressive decoding tasks are forcing instances to remain active for extended lifespan, yet the instances' occupied resources are being underutilized due to shrinking batch scales [27].

To further validate this impact, we conduct an experiment that simulates a 600-second workload transition from high to low RPS traffic, using *decode phase* generation token distributions from Azure trace data. We deploy a set of *decode phase* function instances across 8 GPUs, with each instance configured to satisfy the TPOT SLO constraint under a fixed SM utilization, memory quota, and batch size. The experiment results are visualized in Fig. 1. For the purpose of this analysis, we define an *underutilized instance* as one where the running batch size is less than half of its maximum SLO-compliant batch size (i.e., below 8 for our configured instances).

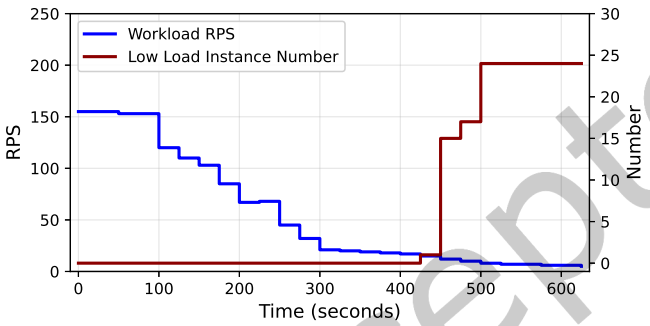


Fig. 1. Number of Low Load *decode phase* Instances vs. Workload Loads Under Low-Traffic Conditions

The experiments show the direct correlation between workload transition and instance underutilization under continuous batching in serverless environments: as traffic drops from high to low RPS at approximately 100 seconds, the number of underutilized *decode phase* instances rises sharply between 400 and 500 seconds, validating that instances are forced to remain active to complete long-sequence decoding tasks. By 500 seconds, all *decode phase* instances enter a persistent low-load state, with no instances operating at their maximum SLO-compliant batch size.

Challenge II

In dynamic load scenarios, continuous batching's decode phase *scheduling clashes with serverless's timely resource release demand: traffic downturns directly prolong the lifespan of underutilized decode instances, and serverless's occupancy billing exacerbates the issue, undermining the paradigm's core goal of resource efficiency.*

3 *CELLServe* Design at a Glance

In this section, we present the design of *CELLServe*. *CELLServe* is positioned as an SLO-aware and cost-efficient serverless platform to optimize the system performance for LLM inference.

As shown in Fig.2, *CELLServe* consists of five core components: offline profiler, scheduler, instance monitor, auto-scaling engine and a gateway handling user request ingress. Before a function is deployed, the offline profiler evaluates the model under different batch sizes and SM allocations, storing phase-specific entries of the form (batch size, SM allocation, latency, memory footprint), where latency is TTFT for *prefill phase* and TPOT for *decode phase* ❶. The profiled performance results ❷ and predefined SLO latency (including TTFT and TPOT requirements) ❸ are transmitted to the auto-scaling engine for subsequent decision-making.

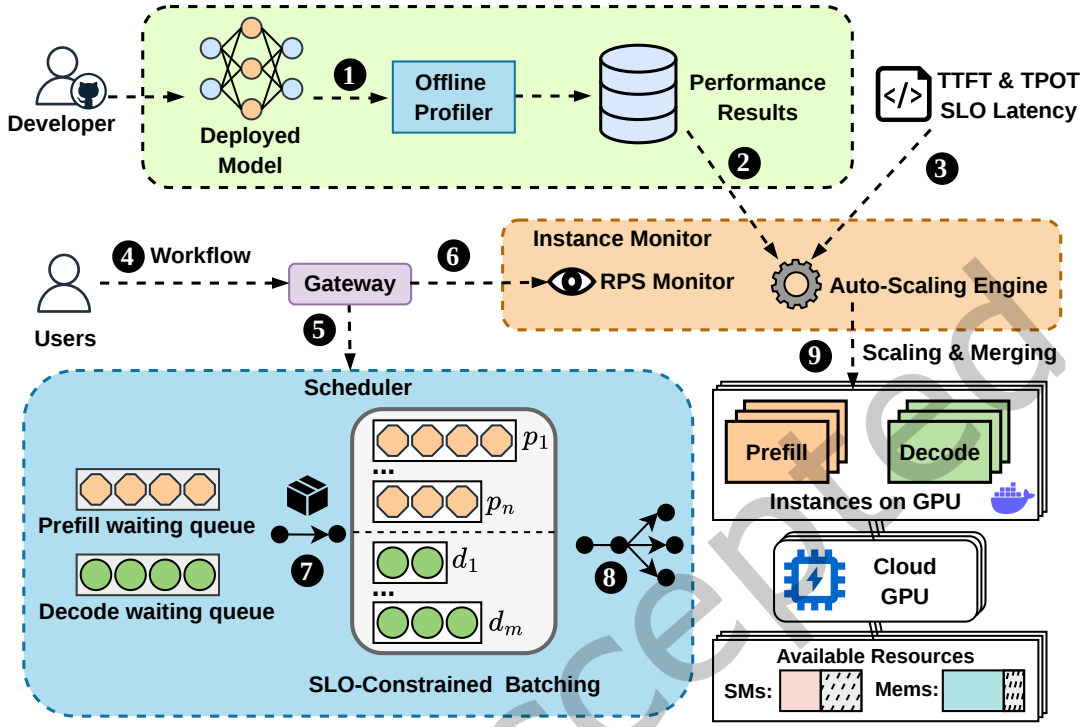


Fig. 2. Architecture of *CELLServe*

On the user side, inference queries are first sent to the gateway ❹, which forwards the requests to both the scheduler and the RPS (Requests Per Second) monitor within the instance monitor ❺. Requests in the scheduler are organized into two separate waiting queues (prefill waiting queue and decode waiting queue) and automatically batched via SLO-constrained batching ❼, which references the profiled performance results, then dispatched to corresponding GPU instances ❽ (equipped with multiple GPUs as available resources). After a prefill instance completes a batch, the scheduler inserts the corresponding requests and first generated tokens into the decode waiting queue for subsequent autoregressive generation, while the selected decode instance pulls the required KV Cache on demand. This coordination links the two phases and allows prefill and decode to scale independently while preserving per-request execution state. The RPS monitor tracks the current system throughput, while the auto-scaling engine compares it with the system's optimal RPS. Based on the pre-profiled model performance results and SLO latency constraints, the auto-scaling engine dynamically scales up/down or merges active instances ❾ to balance SLO compliance and resource efficiency. The auto-scaling engine ❾ serves as the core control component of *CELLServe*, responsible for scaling instances up and down, merging under-utilized *decode phase* instances, and provisioning resources for newly launched instances.

4 Resource Provisioning

To address insufficient compute and memory utilization caused by the asymmetric resource demands between *prefill phase* and *decode phase*, we propose a phase-aware adaptive resource provisioning algorithm for joint resource provisioning. Specifically, we first formulate an optimization problem that aims to minimize the execution cost under available resource and user-defined SLO constraints. Based on this formulation, we design a phase-aware adaptive resource provisioning algorithm that balances performance and cost while ensuring compliance with SLO constraints.

4.1 Problem Analysis And Modeling

CELLServe determines the optimal resource configuration for each instance by formulating and solving an optimization problem that jointly minimizes execution cost while satisfying SLO latency and available resource constraints. In this work, we primarily focus on GPU-related resources and assign a fixed amount of CPU-related resources to each instance, as GPU dominate both the cost structure and the performance bottlenecks in LLM inference. For a given instance i, we define its resource profile as a 3-tuple $\langle b_i, g_i, m_i \rangle$, where b_i is the batch size assigned to instance i, serving as a flexible parameter that adapts to fluctuations in request-per-second (RPS); g_i represents the GPU compute resources allocated to instance i; and m_i denotes the GPU memory resources allocated to instance i. In our implementation based on NVIDIA Multi-Process Service (MPS), g_i corresponds to the percentage of active GPU threads allocated to the instance, and m_i represents the amount of GPU memory reserved. Since LLM inference mainly relies on GPU resources and GPU resources billing is usually much higher than CPU resources, we simplify the CPU-related resource cost by introducing a constant α, representing the fixed amount of CPU-related resources assigned to each instance. Based on the resource profile of instance i, the execution cost is defined as:

$$cost_i = t_i \cdot (g_i \cdot unit_g + m_i \cdot unit_m + \alpha) \tag{1}$$

where $unit_g$ and $unit_m$ represent the unit price of GPU compute and memory resources, respectively; t_i is the actual runtime of instance i.

To determine when to provision new instances, *CELLServe* continuously monitors the system's incoming RPS rate. Provisioning decisions are event-driven, a scale-up action is triggered when the current RPS exceeds the aggregate processing capacity of all active *prefill phase* instances. Formally, scaling is initiated when the following condition is met:

$$R \geq \frac{\sum_{i=1}^{N_p} b_i}{\max_i\{t_i^p\}},\ i \in \{1, \ldots, N_p\} \tag{2}$$

where R denotes the current RPS, N_p denotes the number of active *prefill phase* instances, and t_i^p represents the TTFT time of *prefill phase* instance i. This inequality characterizes the maximum sustainable throughput the current system configuration can support. Let b^p and b^d denote total needed batch size for each phase, N_d denotes the number of active *decode phase* instances and $\hat{b_j}$ denotes the available batch size for *decode phase* instance j, we have the additional required capacity as:

$$b^p = \max_i\{t_i^p\} \cdot R - \sum_{i=1}^{N_p} b_i,\ i \in \{1, ..., N_p\} \tag{3}$$

$$b^d = \max_i\{t_i^d\} \cdot R - \sum_{j}^{N_d} \hat{b_j},\ j \in \{1, ..., N_d\} \tag{4}$$

We consider a server with k available GPUs and seek to determine the optimal configurations for launching new n_p *prefill phase* instances and n_d *decode phase* instances. The decision variables include the resource profile

$\langle b_i, g_i, m_i \rangle$ and an auxiliary binary variable y_{ij}, which is set to "1" if instance i is deployed on GPU j, and "0" otherwise. In addition, we use p_i to indicate that instance i performs the *prefill phase*, and d_i to indicate that it performs the *decode phase*. Hence, the optimization problem can be formulated as follows:

$$\min \sum_{s \in \{p,d\}} \sum_{i=1}^{n_s} t_i^s \cdot (g_i \cdot unit_g + m_i \cdot unit_m + \alpha) \tag{5}$$

$$f_p(b_i, g_i) \mapsto t_i^p, m_i \tag{6}$$

$$f_d(b_i, g_i) \mapsto t_i^d, m_i \tag{7}$$

$$\hat{m}_i \geq m_i \tag{8}$$

$$t_i^p \leq t_{ttft}^{slo}, \ \forall i \in \{1, ..., n_p\} \tag{9}$$

$$t_i^d \leq t_{tpot}^{slo}, \ \forall i \in \{1, ..., n_d\} \tag{10}$$

$$\sum_i^{n_p} g_i^p y_{ij} + \sum_i^{n_d} g_i^d y_{ij} \leq G_j, \ j \in \{1, .., k\} \tag{11}$$

$$\sum_i^{n_p} m_i^p y_{ij} + \sum_i^{n_d} m_i^d y_{ij} \leq M_j, \ j \in \{1, .., k\} \tag{12}$$

$$\sum_{j=1}^{k} y_{ij} = 1 \tag{13}$$

$$\sum_i^{n_p} b_i^p \geq b^p \tag{14}$$

$$\sum_i^{n_d} b_i^d \geq b^d \tag{15}$$

Since the actual runtime of instance i in Eq. (1) is difficult to predict, we replace t_i with t_i^p and t_i^d in the objective (5) as the TTFT for p_i and the TPOT for d_i. The value of t_i^p and t_i^d is obtained via offline profiling based on the resource profile and instance type of instance i, as indicated in constraint (6) and (7). Constraint (8) ensures that the actual provisioned GPU memory for instance i is no less than the profiled usage, to avoid potential runtime errors caused by memory under-provisioning. This substitution makes the objective function more aligned with SLO requirements by replacing the generic runtime metric with phase-specific metrics TTFT and TPOT that directly capture latency constraints in each phase. Constraints (9) and (10) guarantee p_i and d_i satisfy SLO constraints. Constraints (11) and (12) ensure that the allocated compute and memory resources on each GPU do not exceed its capacity, where G_j and M_j denote the total available compute and memory resources on GPU j, respectively. Constraint (13) guarantees that each instance is assigned to exactly one GPU and that its compute and memory allocations are co-located on the same device. Constraints (14) and (15) ensure that launched instances can handle the current RPS rate based on Eq. (3) and (4).

By setting the optimization objective to minimize execution cost in objective (5), *CELLServe* inherently enables phase-aware resource provisioning. Specifically, the algorithm provisions sufficient compute resources to *prefill phase* instances to ensure SLO compliance while avoiding excessive memory provisioning, whereas *decode phase* instances are provisioned with sufficient memory resources with minimizing unnecessary compute resources. This adaptive approach dynamically eliminates resource redundancy by aligning allocations with phase-specific computational and memory bottlenecks.

4.2 Resource Provisioning Algorithm

Ideally, resource provisioning is determined by solving the optimization problem formulated in Eq. (5)–(15). However, this optimization problem is at least as hard as the known NP-hard bin packing problem, due to the combinatorial nature of assigning multiple resource-demanding instances to discrete GPUs under a series of constraints. To prove this, consider a simplified decision version of our problem in which we only keep one inference phase, fix the candidate instance profiles, and ask whether there exists a placement that satisfies the required aggregate batch capacity under the per-GPU compute and memory constraints. In this reduced form, each candidate profile can be viewed as an item with two-dimensional resource demand (g_i, m_i) and capacity contribution b_i, while each GPU is a bin with capacity (G_j, M_j). The problem therefore reduces to a multidimensional bin-packing variant, which is NP-hard. Since our original formulation further includes joint *prefill phase*/*decode phase* provisioning, SLO constraints, and cost minimization, it is at least as hard as this reduced problem. Therefore, instead of solving the exact optimization online, *CELLServe* adopts a greedy heuristic to derive fast SLO-feasible provisioning decisions. Because the underlying problem is NP-hard, Algorithm 1 does not guarantee the global optimum. Instead, it efficiently constructs a feasible provisioning plan under the current resource and SLO constraints.

Algorithm 1 shows the details of *CELLServe*'s resource provisioning algorithm. The algorithm takes the current RPS, latency SLOs for both TTFT and TPOT, and the available compute and memory resources across GPUs as input. The provisioning proceeds in *prefill phase* and *decode phase* respectively. For each phase, the algorithm first computes the required aggregate batch capacity from the real-time RPS using Eq. (3) and Eq. (4) (Line 1). It then scans the offline-profiled candidate resource profiles in descending batch-size order and selects the first feasible profile $\langle b_i, g_i, m_i \rangle$ whose latency satisfies the phase-specific SLO and whose resource demand fits the remaining GPU budget (Lines 2,3). Therefore, the chosen b_i is the largest feasible batch size among the offline-profiled candidate configurations for that provisioned instance under the current resource and SLO constraints. For *prefill phase* provisioning (Lines 4-14), compute-intensive configurations are prioritized and assigned to GPUs with the highest remaining compute capacity. Conversely, memory-intensive configurations for *decode phase* are placed on GPUs with the most available memory (Lines 15-25). Once an instance's resource profile is selected, its footprint is deducted from the server's available resource pool Lines 10-11, 21-22, and the portion of RPS it can serve is subtracted from the remaining demand Lines 12,23. This procedure is repeated until the provisioning requirements for both *prefill phase* and *decode phase* are met, or no feasible configurations remain.

Complexity analysis. Let k be the number of GPUs, let $|C_p|$ and $|C_d|$ denote the numbers of offline-profiled candidate configurations for *prefill phase* and *decode phase*, and let n_p and n_d denote the numbers of newly launched *prefill phase* and *decode phase* instances. Algorithm 1 first sorts GPUs by remaining compute and memory resources, which costs $O(k \log k)$. It then scans candidate profiles for each newly launched instance until finding a feasible one, resulting in worst-case time $O(n_p|C_p| + n_d|C_d|)$. Therefore, the overall time complexity is $O(k \log k + n_p|C_p| + n_d|C_d|)$. Since the candidate profile sets are generated offline and bounded in practice, the online overhead of provisioning remains low.

5 Workload-Aware Instance Manager Under Continuous Batching

To mitigate resource underutilization caused by continuous batching on serverless platforms, *CELLServe* employs an opportunistic instance merging strategy to consolidate low-utilization instances, paired with an optimized KV Cache migration strategy to minimize associated latency overhead and preserve throughput gains.

5.1 Opportunistic Instance Merging Under Low-load

The goal of the instance merging algorithm is to reclaim fragmented GPU resources across low-load instances without triggering inference interruptions or incurring large-scale KV Cache recomputation. To achieve these,

Algorithm 1: ResourceProvisioning(R, $\mathbf{G}$, $\mathbf{M}$, t_{slo})

Input: R: current RPS; $\mathbf{G}$: available GPU compute; $\mathbf{M}$: available GPU memory; t_{slo}: TTFT/TPOT SLOs
Output: n_p, n_d: numbers of prefill/decode instances; $\mathcal{P}, \mathcal{D}$: resource profiles; y_{ij}: placement

$b^p, b^d \leftarrow$ ComputeBatchSizes(R, t_{slo}) // Eq. 3 and Eq. 4
$\mathcal{S}_p \leftarrow$ sort $\mathbf{G}$ by compute descending
$\mathcal{S}_d \leftarrow$ sort $\mathbf{M}$ by memory descending
$\mathcal{P} \leftarrow [\,], n_p \leftarrow 0$
foreach *GPU* $j \in \mathcal{S}_p$ **do**
    while $b^p > 0$ *and GPU* j *has enough resource* **do**
        // first feasible profiled config
        assign b_i, g_i, m_i
        $p_i = \langle b_i, g_i, m_i \rangle$
        $y_{ij} \leftarrow 1$ // allocate prefill instance i on j
        $G_j \leftarrow G_j - g_i$
        $M_j \leftarrow M_j - m_i$
        $b^p \leftarrow b^p - b_i$
        $\mathcal{P}$.append(p_i)
        $n_p \leftarrow n_p + 1$
$\mathcal{D} \leftarrow [\,], n_d \leftarrow 0$
foreach *GPU* $j \in \mathcal{S}_d$ **do**
    while $b^d > 0$ *and GPU* j *has enough resource* **do**
        // first feasible profiled config
        $y_{ij} \leftarrow 1$ // allocate decode instance i on j
        assign b_i, g_i, m_i
        $d_i = \langle b_i, g_i, m_i \rangle$
        $G_j \leftarrow G_j - g_i$
        $M_j \leftarrow M_j - m_i$
        $b^d \leftarrow b^d - b_i$
        $\mathcal{D}$.append(d_i)
        $n_d \leftarrow n_d + 1$
return $n_p, n_d, \mathcal{P} \cup \mathcal{D}, y_{ij}$

CELLServe selectively merges running instances whose GPU utilization falls below a configurable threshold θ. Namely, when multiple instances have:

$$\frac{b_{\text{run}}}{b_{\text{opt}}} \leq \theta,\ \theta \in (0, 1], \tag{16}$$

where b_{run} denotes the actual running batch size and b_{opt} represents the optimal batch size of the target decode instance, i.e., the b_i value of the resource profile selected by Algorithm 1. Merging is also event-driven, once Eq. (16) holds after a decode-state update, the scheduler starts an instance merging process.

Suppose there exists a set of running decode instances whose GPU resource utilization falls below predefined threshold θ according to Eq. (16). For simplicity, we set $\theta = 0.5$ in our implementation. These instances are considered merge candidates. Algorithm 2 shows the details of the merging strategy. In Lines 1–3, we collect all such candidates into a set $\mathbf{I}_{low}$ and construct two heaps over this set. To efficiently pair instances for merging, we sort this set in two different ways and organize them into two heaps: (i) Heap$_A$ (Line 2), maintained as a max-heap

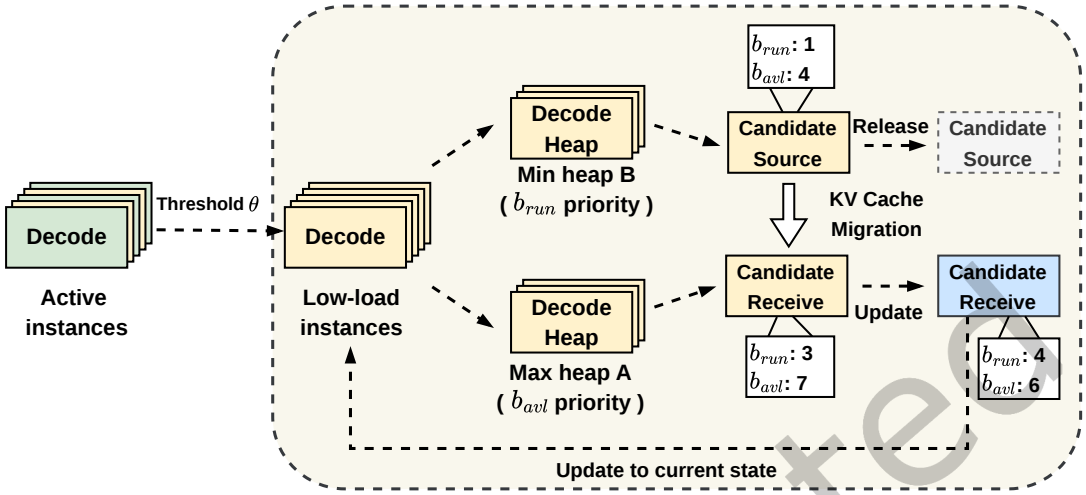


Fig. 3. Example of Instance Merging

keyed by available batch capacity $b_{avl} = b_{opt} - b_{run}$, and (ii) Heap_B (Line 3), maintained as a min-heap keyed by the current running batch size b_{run}. The philosophy of these two heaps is to prioritize merging smaller workloads (low b_{run}) into instances with greater available batch capacity (high b_{avl}), thereby minimizing migration overhead. These two orderings are not generally identical because b_{avl} depends on both b_{run} and the instance-specific target batch size b_{opt} selected by Algorithm 1. Even when multiple instances share the same b_{opt} and the rankings partially coincide, the algorithm still behaves correctly because the chosen receiver is removed from Heap_B before a source instance is selected.

The core merging loop (Lines 4–13) attempts to pop one receive instance d_{recv} from Heap_A and remove the same instance from Heap_B to avoid self-merging. It then pops source instances d_{src} from Heap_B in ascending order of b_{run}. If d_{recv} has sufficient available batch capacity to accommodate the selected source workload, the merging is performed. The actual merging logic involves transferring the active requests and internal state of d_{src} to d_{recv} and updating both the batch size and memory usage of d_{recv} accordingly (Lines 9,10). Once merged, d_{src} is released. If the receive instance no longer has room for further merging, the inner loop breaks and the algorithm continues with the next receiver instance from Heap_A.

Complexity analysis. Let $m = |\mathbf{I}_{low}|$ denote the number of low-utilization *decode phase* instances. Constructing the two heaps costs $O(m \log m)$, and the subsequent heap operations across the merging loop also require $O(m \log m)$ time in total. Therefore, the overall time complexity of Algorithm 2 is $O(m \log m)$, with memory complexity $O(m)$.

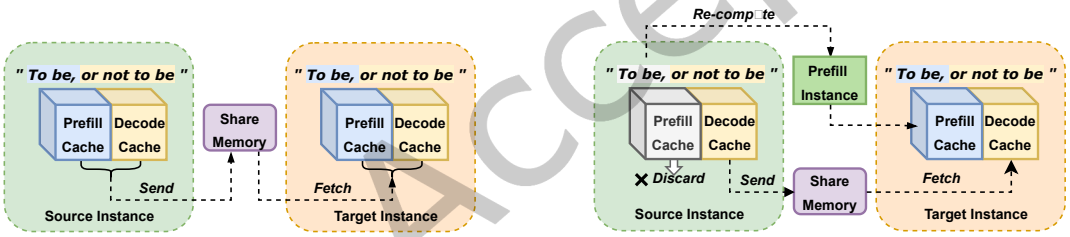
For example, as shown in Fig. 3, low-load *decode phase* instances are filtered by threshold θ, and two decode heaps are built from them: Heap_A (a b_{avail}-prioritized max-heap) and Heap_B (a b_{run}-prioritized min-heap). Suppose Heap_A pops a receive candidate d_{recv} (i.e., the *Candidate Receive* in the figure) with initial $b_{\text{run}} = 3$ and $b_{\text{avail}} = 7$, while Heap_B pops a source candidate d_{src} (i.e., the *Candidate Source* in the figure) with $b_{\text{run}} = 1$ and $b_{\text{avail}} = 4$. Since $d_{\text{recv}} \neq d_{\text{src}}$, and d_{recv}'s $b_{\text{avail}} = 7$ is sufficient to accommodate d_{src}'s $b_{\text{run}} = 1$, the merging process is triggered.

This process starts with the source candidate d_{src} releasing its current workload, followed by KV Cache migration(Section 5.2) that transfers d_{src}'s active requests' KV Cache to the receive candidate d_{recv}. After migration,

Algorithm 2: Opportunistic Instance Merging ($\mathbf{I}_d$, θ)

Input: $\mathbf{I}_d$: active *decode phase* instances; θ: merging threshold
Output: Updated $\mathbf{I}_d$

$\mathbf{I}_{low} \leftarrow \{d_i \in \mathbf{I}_d \mid \text{utilization}(d_i) < \theta\}$; // Identify merge candidates by Eq.(16)
$\text{Heap}_A \leftarrow$ Max heap of $\mathbf{I}_{low}$ with priority of b_{avl}
$\text{Heap}_B \leftarrow$ Min heap of $\mathbf{I}_{low}$ with priority of b_{run}
// Merging instances from Heap B into Heap A
while $Heap_A \neq \emptyset$ ***and*** $Heap_B \neq \emptyset$ **do**
  $d_{recv} \leftarrow \text{Heap}_A$.pop();
  $\text{Heap}_B.pop(\text{d}_{recv})$; ; // Remove d_{recv} to avoid self-merging
  while $Heap_B \neq \emptyset$ **do**
    $d_{src} \leftarrow \text{Heap}_B$.pop(); **if** $b_{avl}^{recv} \geq b_{run}^{src}$ **then**
      $d_{recv} \leftarrow d_{src} + d_{recv}$; // Merge d_{src} and d_{recv}
      $b_{avl}^{recv} \leftarrow b_{avl}^{recv} - b_{run}^{src}$; // Subtract consumed workload from the available batch capacity
    else
      break ; // No more space in d_{recv}
return Updated $\mathbf{I}_d$



(a) Full Transfer (b) Partial Transfer

Fig. 4. KV Cache Migration Strategy

d_{recv}'s state is updated and d_{src}'s resource is released. If the updated receive candidate d_{recv} still has available batch capacity, the algorithm would continue popping source candidates from Heap_B; otherwise, the inner loop breaks, and the next receive candidate is fetched from Heap_A to repeat the merging attempt.

5.2 KV Cache Migration to Avoid Large Scale Recomputation

To reduce overhead of large scale recomputation of KV Cache, *CELLServe* employs an optimized asynchronous KV Cache transmission strategy between *prefill phase* and *decode phase* instances. As shown in Fig. 4(a), full transfer strategy transfers both *prefill phase* and *decode phase* KV Cache from d_{src} to d_{recv}, enabling inference to resume without any recomputation. In contrast, partial transfer strategy, as shown in Fig. 4(b), only transfers the *decode phase* KV Cache, while recomputing the *prefill phase* using available *prefill phase* instances, which enables cache transmission and recomputation to proceed in parallel. Let KV_p and KV_d denote the sizes of the *prefill phase* and *decode phase* KV Caches, respectively, and let bw represent the effective CPU–GPU transfer bandwidth.

Define t_i^p as the TTFT of a candidate *prefill phase* instance p_i. The estimated latency of each migration strategy is given by:

$$t_{\text{full}} = 2 \cdot \frac{KV_p + KV_d}{bw},\ t_{\text{partial}} = \max\{2 \cdot \frac{KV_d}{bw},\ t_i^p\} \tag{17}$$

To further optimize t_i^p in the partial strategy, the system parallelizes the *prefill phase* recomputation across multiple prefill instances by partitioning the KV Cache by request, thereby reducing the overall recomputation latency. If the scheduler identifies that such recomputation can complete faster than full KV Cache transmission, it adopts the partial transfer strategy. Formally, if there exists an available *prefill phase* instance p_i such that

$$t_i^p < 2 \cdot \frac{KV_p + KV_d}{bw}, \tag{18}$$

partial transfer is selected; otherwise, the full transfer strategy is adopted. Such method offers two core advantages for optimized resource utilization and system efficiency. First, it accelerates resource release of d_{src}: the partial transfer strategy offloads only KV_d while paralleling *prefill phase* recomputation, avoiding prolonged resource occupation from full $KV_p + KV_d$ transmission and enabling d_{src} to quickly free up resources for subsequent tasks. Second, it alleviates bandwidth pressure: by reducing transmission volume from $KV_p + KV_d$ to KV_d, the partial strategy cuts data load proportionally to KV_p, mitigating network congestion in distributed systems and freeing up bandwidth for concurrent transfers. Together, these benefits balance migration latency and resource usage, achieving efficient KV Cache migration and fast resource recycling.

6 Technical Implementation

We implement *CELLServe* as a 7,492-line containerized prototype that materializes *prefill phase* and *decode phase* as separate runtime stages. The system follows a control-plane/data-plane architecture. The control plane includes a Proxy, a scheduler, an instance monitor, and an auto-scaling engine, while the data plane consists of phase-specific worker containers built on top of HuggingFace `transformers`. The Proxy handles request ingress; the scheduler maintains separate *prefill phase*/*decode phase* queues, tracks worker availability, and dispatches requests under resource budgets; and the auto-scaling engine serves as the core control component for instance lifecycle management, including scale-up/down, merging under-utilized *decode phase* instances, and provisioning resources for new instances.

At the execution layer, each worker runs as an independent Docker container with a Python inference process. The standard end-to-end generation interface is split into two explicit loops. A *prefill phase* worker processes prompt tokens and materializes the KV tensors needed for subsequent decoding. A *decode phase* worker runs a step-wise autoregressive loop with continuous batching, where finished sequences are removed and queued or migrated requests are admitted whenever local token and memory budgets allow. This design makes phase-specific workers the basic units of placement, scheduling, and reclamation.

The KV handoff follows an inter-process communication (IPC) shared-memory (SHM) design. After *prefill phase*, a worker exports the KV tensors through a shared-memory-based IPC path and sends only metadata to the scheduler, including tensor shapes, sizes, placement information, and the SHM handle. The scheduler forwards this metadata to the selected *decode phase* worker, which imports the referenced KV state and resumes decoding without prompt recomputation. In this way, the control plane transfers only lightweight metadata, while the KV payload itself is handed off out of band.

GPU context sharing is realized with NVIDIA MPS. Each worker runs as an independent CUDA process, and its compute share is controlled through a per-worker active-thread percentage, allowing multiple instances concurrently sharing the same CUDA context on one GPU. GPU memory limits are implemented through PyTorch per-process CUDA memory caps. Together, these mechanisms allow multiple phase-specific workers to be colocated on a single GPU while retaining explicit per-worker control over compute and memory resources.

7 Performance Evaluation

7.1 Experiments Setup

Experimental Platform. We deploy *CELLServe* on a single machine equipped with 8 NVIDIA RTX 4090 GPUs, each with 48GiB of memory. Table 2 summarizes the configuration of the system. We launch all inference functions inside Docker containers, following the containerized deployment model commonly used in serverless platforms.

Table 2. Experimental Testbed Configuration

Component	Specification	Component	Specification
CPU device	AMD EPYC 7K62	Shared LLC Size	384 MiB
Number of sockets	2	CPU Threads	96 (96 physical cores)
Processor BaseFreq.	1.50 GHz	GPU device	NVIDIA GeForce RTX 4090
GPU Memory Config	48 GiB GDDR6X	CUDA cores	16384
Number of GPUs	8	Operating System	Ubuntu 22.04

Model setup. We evaluate *CELLServe* with five representative LLMs: LLaMA2-7B [34], Mistral-7B [17], GLM4-9B [10], Qwen3-14B [40] and Qwen3-0.6B [40]. We select these models to cover a broad range of serving regimes that influence prefill–decode behavior. LLaMA2-7B, Mistral-7B, and GLM4-9B provide representative mid-scale workloads with different latency and throughput characteristics. Qwen3-14B introduces a larger and more resource-demanding setting, which stresses GPU memory capacity and compute provisioning, while Qwen3-0.6B represents a lightweight setting with lower per-request cost and substantially higher concurrency potential. This combination enables us to evaluate *CELLServe* under both heavy-resource and high-concurrency scenarios, rather than focusing on a single model scale or workload profile.

We empirically determine the SLO configurations according to each service's target performance, as no publicly available SLO specifications exist for these deployments to the best of our knowledge. The SLO targets are chosen based on offline profiling of the model-specific latency characteristics on our testbed, and are then applied uniformly to all systems.

Workload setup. For workload generation, we adopt the Azure LLM Inference Trace [27] released by Microsoft Azure, a one-hour production trace with real-world requests and timestamps. The dataset natively provides two categories of workloads, *Azure Code Trace* and *Azure Conversation Trace*, each corresponding to a distinct application class captured by Azure's production logging pipeline. The two traces exhibit distinct workload characteristics: the code-generation trace produces fewer output tokens but substantially longer prefilling lengths, while the conversational trace contains more decoding-heavy requests.

For workload scaling, we construct request streams from the trace to match the target RPS required by our experimental setup and comparison settings as the benchmark dataset. Specifically, for an experiment lasting T seconds with a target rate of R requests per second, we require $R \times T$ requests in total. To obtain such a workload, we randomly select a contiguous segment of the trace containing the required number of requests. We then replay these requests after linearly rescaling their timestamps according to the ratio between the target duration T and the original span of the selected segment. In this way, we preserve the relative arrival pattern of the trace segment while matching the desired experiment duration and request rate. We do not apply additional filtering or preprocessing beyond selecting representative load intervals, ensuring that the workload remains faithful to the original trace characteristics.

Metrics. We use throughput as the primary evaluation metric. Under a fixed and reasonable SLO configuration, defined jointly by TTFT and TPOT, we measure the maximum number of tokens the system can generate and the number of requests it can concurrently serve. In addition, we evaluate memory usage and SM utilization when different systems operate at comparable throughput levels.

Baselines. We compare *CELLServe* with three mainstream serverless platforms, OpenWhisk, OpenFaaS and ServerlessLLM [7]: **OpenWhisk**[3] is an open-source, event-driven serverless framework originally developed by IBM. It also serves as a common baseline for serverless inference research. Prior work such as InstaInfer [33] and Infless [41] conducts their system evaluations on top of OpenWhisk. **OpenFaaS**[4], in contrast, is a lightweight, Kubernetes-native platform that encapsulates functions as Docker containers and relies on Kubernetes' autoscaling capabilities to dynamically manage compute resources according to workload intensity. **ServerlessLLM**[8] is the current state-of-the-art serverless inference system specifically designed for large language models. By enabling efficient model reuse across invocations and minimizing redundant data movement, it achieves low-latency and cost-effective LLM inference in serverless environments.

7.2 Overall Performance Evaluation

7.2.1 Maximum Throughput. To evaluate end-to-end serving efficiency, we measure the maximum sustainable throughput of *CELLServe*, OpenWhisk, OpenFaaS and ServerlessLLM under both the *Azure Conversation* and *Azure Code* traces. For each trace, we select five representative load intervals. In the rest of this section, we refer to these five load intervals using qualitative labels *Low*, *Low-Mid*, *Mid*, *Mid-High*, and *High*. For each model, the evaluated load levels are chosen to span the under-utilized, moderately loaded, and near-saturation regimes for all compared systems. The same per-model load settings are applied consistently to *CELLServe* and all baselines.

For the smaller Qwen3-0.6B model, we adopt higher arrival rates to ensure comparable system utilization and stress levels across models. Specifically, the load intervals are {25, 75, 125, 175, 225} req/s for *Azure Conversation* and {150, 350, 550, 750, 950} req/s for *Azure Code*. This adjustment is necessary because Qwen3-0.6B has significantly lower per-request computation cost, and using the same arrival rates as other models would fail to sufficiently stress the system. For all other models, these labels correspond to average arrival rates of {25, 50, 75, 100, 125} req/s for the *Azure Conversation* trace and {50, 150, 250, 350, 450} req/s for the *Azure Code* trace. This design ensures that all systems are evaluated under comparable utilization regimes rather than identical request rates.

To ensure a fair comparison across platforms, we evaluate maximum throughput under consistent latency QoS constraints. We determine a TTFT–TPOT target for each model family based on its achievable performance envelope. Specifically, we set LLaMA2-7B: (TTFT 1500 ms, TPOT 30 ms), GLM4-9B: (TTFT 3000 ms, TPOT 60 ms), Mistral-7B: (TTFT 2500 ms, TPOT 80 ms), Qwen3-14B: (TTFT 3000 ms, TPOT 45 ms), Qwen3-0.6B: (TTFT 250 ms, TPOT 30 ms) .

As shown in Figure 5, under the conversation trace, *CELLServe* outperforms OpenWhisk by 1.09×–1.73×, OpenFaaS by 1.08×–1.64×, and ServerlessLLM by 1.08×–1.52× across all evaluated models. Under the code trace, *CELLServe* further improves over OpenWhisk by 1.23×–1.85×, OpenFaaS by 1.23×–1.71×, and ServerlessLLM by 1.21×–1.78×. The acceleration achieved by *CELLServe* arises from the inherently asymmetric nature of the two traces: prefilling and decoding impose drastically different computational loads. Systems that couple these two stages into a single execution pipeline are therefore prone to queue interference, where the heavier stage delays the lighter one and suppresses overall throughput. By isolating and independently scaling the two pipelines, *CELLServe* mitigates this interference and maintains higher effective GPU utilization. Specifically, Qwen3-14B shows marginal gains and represents a boundary case in our evaluation: its larger memory footprint prevents multiple instances from being colocated on one GPU in our setting, leaving limited room for *CELLServe* to

[3]https://openwhisk.apache.org/
[4]https://www.openfaas.com/

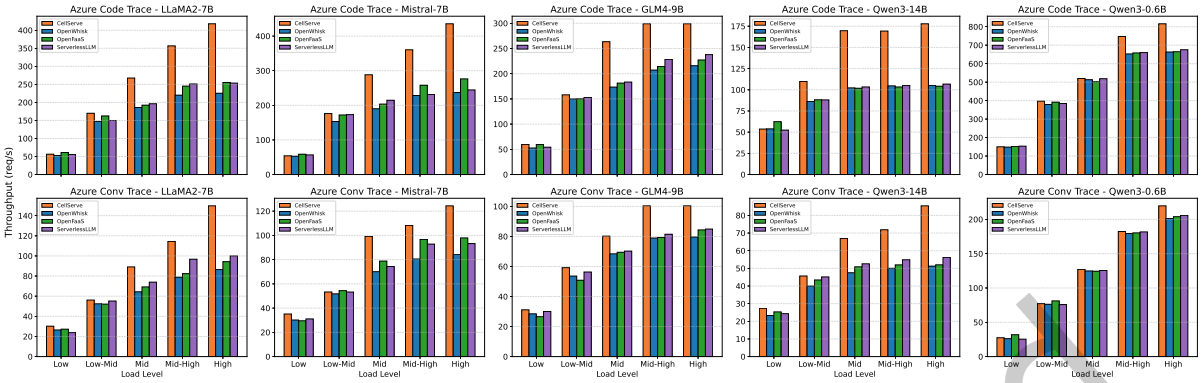


Fig. 5. Maximum throughput comparison of *CELLServe* against OpenWhisk, OpenFaaS, and ServerlessLLM across five load levels under Azure conversation and code traces for LLaMA2-7B, Mistral-7B, GLM4-9B, Qwen3-14B, and Qwen3-0.6B.

improve throughput. Therefore, we focus the subsequent detailed analyses on models where phase-level resource packing and instance merging are feasible.

7.2.2 Latency Breakdown. To further analyze the execution behavior of each platform, we perform a detailed latency breakdown using the same Azure Code Trace segments introduced in Section 7.2.1. Specifically, we replay the five representative load intervals—corresponding to average arrival rates of 50, 150, 250, 350, and 450 req/s—and record the end-to-end latency of every request.

For *CELLServe*, we decompose the latency into four components: (i) the waiting time before a request is admitted to a prefill instance (*P_wait*), (ii) the prefill execution time (*Prefill*), (iii) the waiting time in the decode queue (*D_wait*), and (iv) the decode execution time (*Decode*). For OpenWhisk, OpenFaaS, and ServerlessLLM, which do not support KV Cache migration and thus cannot realize prefill/decode separation, we instrument an analogous breakdown by attributing the time before the first token to the prefill side and the remaining time to the decode side. To ensure a fair comparison, we additionally track the request-level waiting time introduced by the serverless scheduling layer, allowing us to verify whether each configuration satisfies the prescribed latency SLOs and to capture queuing effects that are absent in our design.

Fig. 6 reports the latency breakdown of four models under different request rates. *CELLServe* consistently achieves lower end-to-end latency than all compared systems across most load levels. Specifically, *CELLServe* reduces end-to-end latency by 42.9%–60.7% compared to OpenWhisk, 39.2%–57.8% compared to OpenFaaS, and 34.8%–54.1% compared to ServerlessLLM.

At low request rates, the latency is dominated by the *Prefill* and *Decode* stages, whereas the queueing components, *P_wait* and *D_wait*, remain relatively small. As the load increases, queueing delay becomes the dominant contributor to the overall latency for all platforms. Nevertheless, the growth of queueing delay under *CELLServe* is markedly more moderate for these models. At 450 req/s, the combined *P_wait* and *D_wait* under *CELLServe* reaches 5494.3 ms for LLaMA2-7B, 933.9 ms for Mistral-7B, 1605.7 ms for GLM4-9B and 2767.0 ms for Qwen3-0.6B. In contrast, for the four models with baseline data, the competing platforms exhibit substantially more severe queue buildup at high load, with *P_wait* alone typically increasing to 4.7–7.0 s at 450 req/s. These results indicate that *CELLServe* is more effective at containing queueing delay under heavy traffic, thereby providing lower end-to-end latency across a broad range of serving conditions.

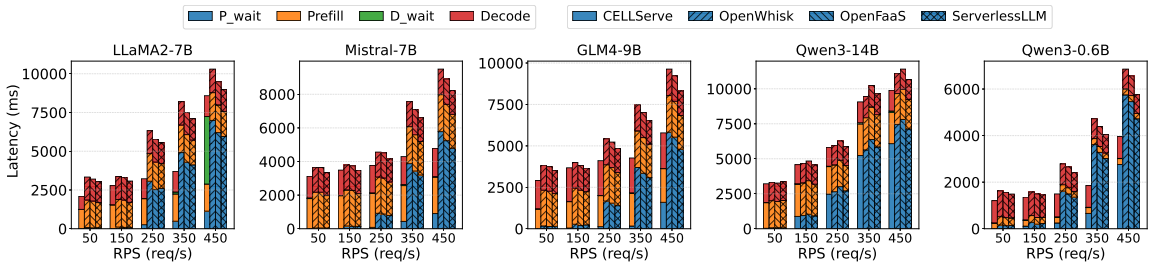


Fig. 6. Latency breakdown across different platforms under varying request rates for LLaMA2-7B, Mistral-7B, GLM4-9B, and Qwen3-0.6B.

7.3 Effectiveness of Prefill–Decode Resource Allocation

We compare several GPU resource *configuration plans* that specify how much capacity is reserved for the *Prefilling* and *Decoding* stages in *CELLServe*. Our goal is to understand how different fixed allocation schemes affect overall throughput and latency under identical SLO requirements.

In this context, a configuration plan is a fixed policy that partitions GPU resources, particularly SMs and memory capacity, between Prefilling and Decoding instances. These predefined ratios determine the effective compute and memory budgets of each stage and directly shape the end-to-end performance. Since the achievable throughput is bounded by the smaller capacity of the two stages, a good configuration should keep Prefilling and Decoding balanced while maximizing total resource utilization.

To assess the impact of different configuration plans, we reuse the same request trace and the same representative RPS levels (ranging from *Low*, *Low–Mid*, *Mid*, *Mid–High*, to *High* load) introduced in Section 7.2.1. Within each selected interval, the device allocation pattern of *CELLServe* is held fixed, and we measure the maximum sustainable throughput under stable high-RPS conditions. The resulting dynamically optimized allocation is then compared against four static baselines:

(1) *SM/Memory-Equal (Symmetric)*: Prefill and Decode receive equal shares of SMs and device memory.
(2) *SM50-50, BS1:16*: Prefill and Decode each obtain 50% of the available SMs, while Decode uses a batch size 16× larger than Prefill.
(3) *SM80-20, BS1:16*: Prefill receives 80% of the SMs and Decode 20%, with Decode batch size 16× larger than Prefill.
(4) *SM20-80, BS1:16*: Prefill receives 20% of the SMs and Decode 80%, again with Decode batch size 16× larger than Prefill.

We measure the system-level throughput under each RPS interval to ensure that all configurations are evaluated under comparable load conditions. Fig. 7 compares the maximum achievable throughput of *CELLServe* under this high-RPS conversational workload with all four static baselines described above. Across both traces and all five models, *CELLServe* consistently achieves substantially higher throughput than the baselines. In terms of peak improvement within each trace–model pair, *CELLServe* delivers up to 1.1×–4.3× higher throughput on the conversational trace and 1.2×–9.2× on the code-generation trace compared to the strongest static configuration. Across the full load spectrum, the throughput advantage remains consistently significant. Specifically, the improvement ranges from 0.9×–4.3× under the conversational trace and 0.9×–9.2× under the code-generation trace across all evaluated models. At low RPS, *CELLServe* intentionally activates fewer GPU resources, so its throughput reflects the minimum resource required to satisfy the SLO rather than the maximum capacity under static allocations.

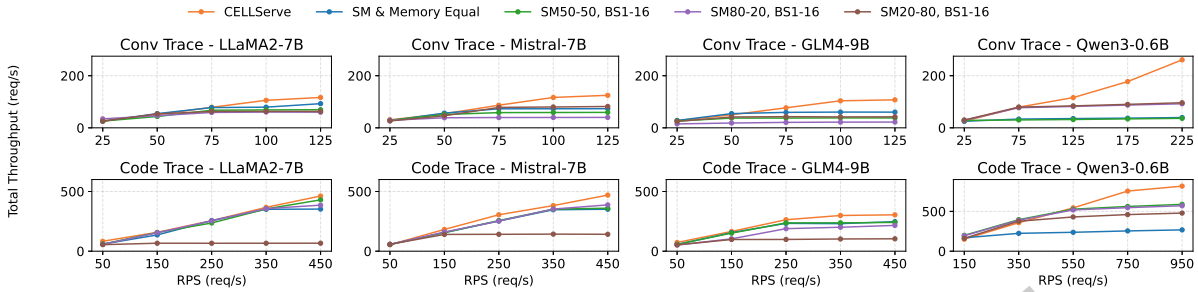


Fig. 7. Throughput comparison under different allocation plan

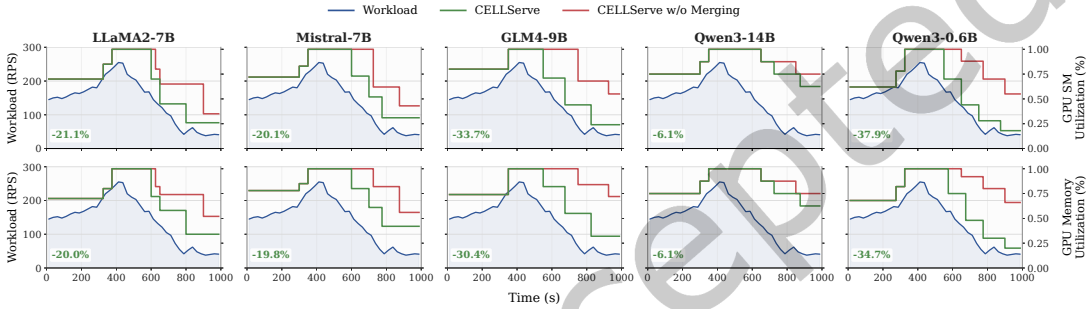


Fig. 8. Resource Comsumption vs Workloads

This performance advantage arises from two fundamental limitations of static deployment. First, fixed allocations inherently create an imbalance between Prefill and Decode phases, leading to asymmetric throughput and unavoidable waiting. Second, static configurations cannot adapt to workload dynamics, leaving substantial GPU resources underutilized—either in the form of idle SMs or unused memory capacity.

7.4 Effectiveness of Opportunistic Instance Merging

To assess the effectiveness of the opportunistic instance merging strategy, we sample thousands of inference requests from Azure code traces, where the workload exhibits a clear trend of transitioning from high load to low load, and we sampled the resource consumption every 25 seconds. The experimental results regarding resource consumption under this workload trend are presented in Fig. 8. To standardize the measurement of resource consumption, we adopt a normalization method in which we take the maximum resource usage within the target time period as the baseline, and normalize resource consumption values accordingly.

We use "$\sum T \times R_T$" (sum of time multiplied by normalized utilization) to measure the resource consumption between different serving modes and calculate data starting from 425 seconds. In terms of SM Utilization, the Merge mode's consumption exhibits an average reduction of 23.8% compared to the No Merge mode: for LLaMA2-7B, it drops from 417.50 to 329.50; for Mistral-7B, from 446.75 to 356.75; for GLM4-9B, from 485.25 to 321.75; for Qwen3-14B, from 512.50 to 481.25; the relatively modest gain on Qwen3-14B is expected, since each instance already occupies an entire GPU in the setting, thus leaving limited gain for further decode-instance merging; and for Qwen3-0.6B, from 477.50 to 296.50. The GPU Memory Utilization shows the same optimization trend, with an average drop of 22.2%: LLaMA2-7B's consumption decreases from 457.50 to 366.00, GLM4-9B's consumption

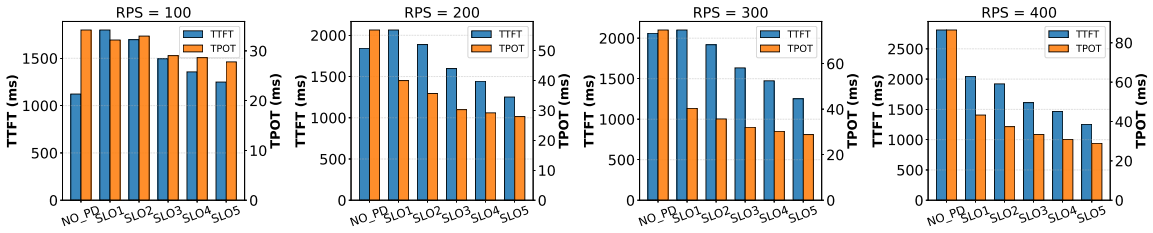


Fig. 9. Throughput comparison under mixed and separate prefill-decode setup

declines from 526.00 to 366.00, and Mistral-7B's consumption reduces from 493.00 to 395.25; for Qwen3-14B, the consumption decreases from 512.50 to 481.25; for Qwen3-0.6B, it drops from 506.00 to 330.50. These results confirm that the merging strategy effectively cuts redundant resource occupancy caused by scattered instances, achieving stable and significant resource-saving effects across different model scales and architectures.

8 Discussion

8.1 Prefill-Decode Separation Analysis

To quantify the impact of Prefill–Decode separation under realistic serving dynamics, we perform a comparison on GLM4-9B using the *Azure Code Trace*. We compare *CELLServe*—which autonomously scales resources under five SLO configurations (SLO1–SLO5)—against a *non–PD (mixed prefill–decode)* baseline. Unlike *CELLServe*, the baseline does not apply SLO-driven scaling; instead, it is provisioned with the same total SM and memory resources as *CELLServe* under each SLO, making the comparison controlled and capacity-equivalent. For reference, *CELLServe* targets TTFT thresholds of 2500/2200/1900/1600/1300ms and TPOT thresholds of 45/40/35/30/25ms for SLO1–SLO5, respectively. Both systems are evaluated under increasing request rates. The resulting TTFT and TPOT curves are shown in Fig. 9.

Empirically, we observe that under low request rates, the TTFT of *CELLServe* is slightly higher than that of the mixed pipeline, and TPOT remains comparable across all SLO levels. In this regime, the system is far from saturation and the benefits of structural decoupling are not yet dominant. As the load increases, however, the trends diverge: the mixed pipeline exhibits rapid TTFT and TPOT inflation, especially under stricter SLOs (SLO3–SLO5), while the Prefill–Decode separated design maintains significantly lower TTFT and TPOT across the same load range.

These observations indicate that Prefill–Decode separation introduces a small overhead at light load, but substantially improves latency once the system enters the medium-to-high utilization regime. By decoupling prefill and decode, *CELLServe* eliminates queue interference between the two stages and mitigates head-of-line blocking, allowing each stage to scale according to its own bottlenecks. As a result, the separated design provides more stable and predictable latency under stringent SLOs, which validates the effectiveness of the Prefill–Decode architecture adopted in *CELLServe*.

8.2 Overhead analysis

We analyze the runtime overhead introduced by *CELLServe* to evaluate whether its architectural design, including Prefill–Decode disaggregation and GPU space partitioning, incurs non-negligible system cost. The overhead mainly arises from interactions between the centralized request handler and the decoupled Prefill and Decode instances, including request scheduling, status updates, and KV cache management.

The major source of system overhead comes from coordination between independent Prefill/Decode processes and the central request handler. Each instance process periodically fetches incoming requests, updates their

execution status, and loads the corresponding KV Cache for decoding. For the LLaMA2-7B model, the measured timings of these operations are as follows: enqueuing a request into the wait for the prefill queue takes 5.2 ms, fetching by the Prefill process takes 0.06 ms, adding completed requests to the wait for decode queue requires 2.8 ms, and fetching by the Decode process costs 2.9 ms.

During computation, the Prefill stage (SM = 100) consumes approximately 0.53 s, while the Decode stage (also SM = 100) takes 11.2 s for generating 200 tokens. The per-token KV Cache load time averages 3.1 ms, and the complete KV Cache transfer and merge ("engine destroy") between stages requires 1.16 s in total for all 64 tokens.

In addition, the system includes an offline profiler and an RPS monitor. The offline profiler measures model performance before deployment, and therefore introduces no overhead to online serving. The RPS monitor records request arrivals and completions in real time, while maintaining these statistics asynchronously and fully in parallel with normal request processing. Therefore, the overhead introduced by monitoring is negligible.

8.3 Cold Start Issues

Cold start affects the responsiveness of serverless LLM serving, especially when new capacity must be activated under bursty demand. In *CELLServe*, however, this issue arises from worker readiness rather than from Prefill–Decode separation itself. The latency of a cold start is mainly determined by instance-lifecycle operations such as model fetching, weight loading, container and runtime initialization, and GPU context setup, all of which take place before steady-state phase execution begins. Prior studies on serverless systems and serverless LLM serving have therefore focused on reducing initialization overhead through warm pools, lazy initialization, and instance reuse [7, 15, 26]. From this perspective, cold start is best understood as a readiness problem that is largely orthogonal to the phase-level interference addressed in this work. A practical extension of *CELLServe* would be to maintain a small reserve of pre-initialized workers and proactively materialize model states during low-load periods, so that capacity expansion can be triggered with lower first-request latency.

8.4 Scalability Discussion

8.4.1 Multi-Model Serving. While our evaluation focuses on single-model serving, the design of *CELLServe* is not inherently limited to that setting. When multiple models are served concurrently over a shared GPU pool, admission, placement, and scaling decisions become coupled across serving pipelines because different models exhibit heterogeneous TTFT/TPOT characteristics, memory footprints, and batching efficiency [44]. The current design of *CELLServe* already provides flexibility for this setting through per-model offline profiling, fine-grained phase-aware resource provisioning, and opportunistic instance merging, a model-agnostic mechanism for reclaiming under-utilized capacity. However, fully supporting multi-model serving, would require model-aware scheduling that jointly accounts for cross-model interference, fairness, and model-level priority.

8.4.2 Multi-Node Deployment. For multi-node deployment, the key challenge is efficient cross-node KV-cache transfer under heterogeneous communication support [22, 27, 48]. This makes routing and scheduling substantially more demanding, as the system must jointly reason about GPU availability, topology, locality, and communication overhead. While the phase-specific instance abstraction of *CELLServe* can naturally scale over a larger GPU pool, preserving the benefit of *prefill phase–decode phase* disaggregation requires topology-aware routing and scheduling that prioritizes same-node placement and falls back to cross-node execution only when local capacity is insufficient.

8.4.3 Mixture-of-Experts Models. Compared with the dense models considered in this work, Mixture-of-Experts (MoE) models introduce additional deployment challenges because they often require larger memory footprints and expert parallelism (EP) to shard expert weights across GPUs, creating extra placement and communication constraints. In this setting, the performance of *prefill phase* and *decode phase* is shaped not only by phase

asymmetry, but also by sparse routing, expert imbalance, and expert-parallel communication [11, 49]. The current design of *CELLServe* provides a foundation for supporting MoE models, since its provisioning decisions are driven by offline profiling of TTFT, TPOT, and memory footprint under different resource configurations. Efficient support for MoE models, however, would require the profiling and scheduling pipeline to explicitly account for EP-related behavior, including routing skew, expert activation frequency, communication volume, and expert locality.

9 Related Work

LLM serving systems. With the popularity of transformer-based LLMs, extensive research has centered on optimizing LLM serving. Orca [42] introduces continuous batching to reduce request latency and improve GPU utilization; FlexGen [31] optimizes throughput for LLM inference in resource-constrained offline scenarios, making it unsuitable for online serving; FastServe [37] proposes a preemptive scheduling framework to minimize LLM inference job completion times; vLLM [21] employs the PagedAttention mechanism to efficiently manage KV caches. Research has also extended to serverless frameworks for LLM inference. ServerlessLLM [8], DeepServe [15], and FaaSwap [43] primarily improve serverless LLM inference through cold-start mitigation, model-loading optimization, and runtime engineering; Medusa [5] reduces cold start through offline state materialization of CUDA graphs; HydraServe [26] cuts cold start latency via proactive model distribution and stage overlapping. However, these systems do not explicitly address the two serverless-specific problems targeted by CELLServe: SLO-aware joint provisioning for independently scaled *prefill phase*/*decode phase* instances, and instance lifespan management for long-lived underutilized decode instances under continuous batching.

Disaggregated LLM Inference. To eliminate *phase interference*, recent systems adopt PD disaggregation across heterogeneous hardware clusters. DistServe [48] assigns *prefill* and *decode phase* to separate GPUs to improve the system's performance; Splitwise [27] explores both homogeneous and heterogeneous device configurations to balance cost, throughput, and power efficiency for PD disaggregation deployment; semi-PD [13] further studies phase-wise disaggregated LLM serving by combining disaggregated computation with unified storage, highlighting the importance of efficient coordination between the two phases; TetriInfer [14] adopts a two-level scheduling algorithm to balance GPU loads under PD disaggregation. Beyond PD disaggregation, systems like Sarathi-serve [2] further optimize the inference pipeline by introducing chunked prefill mechanisms.

Another emerging direction is Attention-FFN (AF) disaggregation in heterogeneous systems, orthogonal to PD disaggregation, which decouples the two core modules of the LLM layers for specialized optimization. MegaScale-Infer [19] targets MoE models by disaggregating attention and FFN modules to boost GPU utilization; Step-3 [35] proposes Attention-FFN disaggregation as part of its hardware-aware co-design, achieving high throughput for long-context reasoning tasks.

Building on the above optimizations for LLM serving, our work focuses on the integration of PD disaggregation and continuous batching in serverless platforms. Unlike prior serverless LLM systems that mainly optimize startup and runtime overheads, and unlike prior disaggregated serving systems that focus on conventional cloud deployments, CELLServe directly addresses the resulting serverless-specific problems of joint provisioning and decode-instance lifespan management. Adopting an SLO-aware design, our research aims to maximize throughput while minimizing costs, which aligns with the core characteristic of serverless computing.

10 Conclusions and Future Work

Serverless has emerged as a promising solution for LLM serving due to its elastic scaling and pay-as-you-go billing model. However, our study shows that existing solutions fail to efficiently combine state-of-the-art LLM inference optimizations with serverless platform, which is non-trivial. Therefore, purpose *CELLServe*, a SLO-aware and cost efficent solution for serverless LLM serving, combined with PD disaggregation and continous batching. Our

evaluations using real-world Azure LLM traces and 5 mainstream models demonstrate that *CELLServe* significantly improves throughput while maintaining strict TTFT and TPOT SLOs, achieving substantial performance gains under identical resource budgets. As part of future work, we plan to further optimize multi-model concurrency and explore more advanced migration strategies to enhance efficiency in heterogeneous serverless environments.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *USENIX Symposium on Operating Systems Design and Implementation*. 117–134.

[3] 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).

[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 *Proceedings of Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 12562–12584.

[5] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774* (2024).

[6] Sumit Kumar Dam, Choong Seon Hong, Yu Qiao, and Chaoning Zhang. 2024. A complete survey on llm-based ai chatbots. *arXiv preprint arXiv:2406.16937* (2024).

[7] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. arXiv:2401.14351 [cs.LG] https://arxiv.org/abs/2401.14351

[8] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM:Low-Latency serverless inference for large language models. In *USENIX Symposium on Operating Systems Design and Implementation*. 135–153.

[9] Lei Gao, Chaoyi Jiang, Hossein Entezari Zarch, Daniel Wong, and Murali Annavaram. 2025. DuetServe: Harmonizing Prefill and Decode for LLM Serving via Adaptive GPU Multiplexing. *arXiv preprint arXiv:2511.04791* (2025).

[10] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793* (2024).

[11] Seokjin Go and Divya Mahajan. 2025. Moetuner: Optimized mixture of expert serving with balanced expert placement and token routing. *arXiv preprint arXiv:2502.06643* (2025).

[12] Jianfeng Gu, Puxuan Wang, Isaac David Núñez Araya, Kai Huang, and Michael Gerndt. 2025. HAS-GPU: Efficient Hybrid Auto-scaling with Fine-Grained GPU Allocation for SLO-Aware Serverless Inferences. In *European Conference on Parallel Processing*. Springer, 159–174.

[13] Ke Hong, Lufang Chen, Zhong Wang, Xiuhong Li, Qiuli Mao, Jianping Ma, Chao Xiong, Guanyu Wu, Buhe Han, Guohao Dai, et al. 2025. semi-pd: Towards efficient llm serving via phase-wise disaggregated computation and unified storage. *arXiv preprint arXiv:2504.19867* (2025).

[14] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. 2024. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181* (2024).

[15] Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Jie Meng, Baoquan Zhang, Shining Wan, et al. 2025. DEEPSERVE: Serverless Large Language Model Serving at Scale. *arXiv preprint arXiv:2501.14417* (2025).

[16] Jinqi Huang, Yi Xiong, Xuebing Yu, Wenjie Huang, Entong Li, Li Zeng, and Xin Chen. 2025. SLO-Aware Scheduling for Large Language Model Inferences. *arXiv preprint arXiv:2504.14966* (2025).

[17] Albert Q. Jiang et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* 3 (2023).

[18] Jiantong Jiang, Peiyu Yang, Rui Zhang, and Feng Liu. 2025. Towards efficient large language model serving: A survey on system-aware kv cache optimization. *Authorea Preprints* (2025).

[19] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *USENIX Symposium on Networked Systems Design and Implementation*. 745–760.

[20] Lingxiao Jin, Zinuo Cai, Haoxin Wang, Zongpu Zhang, Ruhui Ma, Haibing Guan, Yuan Liu, and Rajkumar Buyya. 2025. Ephemera: Accelerating I/O-Intensive Serverless Workloads with a Harvested In-memory File System. *ACM Transactions on Architecture and Code Optimization* (2025).

[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 *Proceedings of Symposium on Operating Systems Principles*. 611–626.
[22] Jiaxi Li, Yue Zhu, Eun Kyung Lee, and Klara Nahrstedt. 2025. Revisiting Disaggregated Large Language Model Serving for Performance and Energy Implications. *arXiv preprint arXiv:2601.08833* (2025).
[23] Liwei Lin, Rongbo Ma, Zejian Wang, Zinuo Cai, Haochen Xu, Baoheng Zhang, Ruhui Ma, and Rajkumar Buyya. 2025. HWDSQP: a Historical Weighted and Dynamic Scheduling Quantum Protocol to Enhance Communication Reliability. *IEEE Journal on Selected Areas in Communications* (2025).
[24] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. 2025. DeepSeek-V3. 2: Pushing the Frontier of Open Large Language Models. *arXiv preprint arXiv:2512.02556* (2025).
[25] Junhan Liu, Zinuo Cai, Yumou Liu, Hao Li, Zongpu Zhang, Ruhui Ma, and Rajkumar Buyya. 2025. SMORE: Enhancing GPU Utilization in Deep Learning Clusters by Serverless-based Co-location Scheduling. *IEEE Transactions on Parallel and Distributed Systems* (2025).
[26] Chiheng Lou, Sheng Qi, Chao Jin, Dapeng Nie, Haoran Yang, Yu Ding, Xuanzhe Liu, and Xin Jin. 2025. HydraServe: Minimizing Cold Start Latency for Serverless LLM Serving in Public Clouds. arXiv:2502.15524 [cs.DC] https://arxiv.org/abs/2502.15524
[27] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *ACM/IEEE Annual International Symposium on Computer Architecture*. IEEE, 118–132.
[28] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In *Proceedings of the International Conference on Intelligent User Interfaces*. 491–514.
[29] Haiying Shen and Tanmoy Sen. 2025. AccelGen: Heterogeneous SLO-Guaranteed High-Throughput LLM Inference Serving for Diverse Applications. *arXiv preprint arXiv:2503.13737* (2025).
[30] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. 2024. Fairness in serving large language models. In *USENIX Symposium on Operating Systems Design and Implementation*. 965–988.
[31] 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*. PMLR, 31094–31116.
[32] Hongjian Shi, Yicheng Di, Xinyu Ruan, Mingrui Liao, Qian Zhang, Ruhui Ma, and Haibing Guan. 2025. Efficient federated recommender system with adaptive model pruning and momentum-based batch adjustment. *ACM Transactions on Recommender Systems* (2025).
[33] 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 *Proceedings of ACM Symposium on Cloud Computing*. 178–195.
[34] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[35] Bin Wang, Bojun Wang, Changyi Wan, Guanzhe Huang, Hanpeng Hu, Haonan Jia, Hao Nie, Mingliang Li, Nuo Chen, Siyu Chen, et al. 2025. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding. *arXiv preprint arXiv:2507.19427* (2025).
[36] Li Wang, Yankai Jiang, and Ningfang Mi. 2024. Advancing serverless computing for scalable ai model inference: Challenges and opportunities. In *Proceedings of the 10th International Workshop on Serverless Computing*. 1–6.
[37] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920* (2023).
[38] Haoyi Xiong, Jiang Bian, Yuchen Li, Xuhong Li, Mengnan Du, Shuaiqiang Wang, Dawei Yin, and Sumi Helal. 2024. When Search Engine Services Meet Large Language Models: Visions and Challenges. *IEEE Transactions on Services Computing* 17, 6 (2024), 4558–4577. doi:10.1109/TSC.2024.3451185
[39] Chuhao Xu, Zijun Li, Quan Chen, Han Zhao, and Minyi Guo. 2025. LLM-Mesh: Enabling Elastic Sharing for Serverless LLM Inference. *arXiv preprint arXiv:2507.00507* (2025).
[40] An Yang et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388
[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 *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 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 *USENIX Symposium on Operating Systems Design and Implementation*. 521–538.
[43] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. 2023. Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping. *arXiv preprint arXiv:2306.03622* (2023).
[44] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, et al. 2025. Prism: Unleashing gpu sharing for cost-efficient multi-llm serving. *arXiv preprint arXiv:2505.04021* (2025).
[45] Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471* (2025).

[46] Shuoming Zhang, Jiacheng Zhao, Siqi Li, Xiyu Shi, Yangyu Zhang, Shuaijiang Li, Donglin Yu, Zheming Yang, Yuan Wen, Huimin Cui, et al. 2025. SpaceServe: Spatial Multiplexing of Complementary Encoders and Decoders for Multimodal LLMs. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[47] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[48] 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 *USENIX Symposium on Operating Systems Design and Implementation*. 193–210.

[49] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, et al. 2025. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism. *arXiv preprint arXiv:2504.02263* (2025).

Received 12 December 2025; revised 2 June 2026; accepted 30 June 2026