EcoServe: Enabling Cost-effective LLM Serving with Proactive Intra- and Inter-Instance Orchestration

Jiangsu Du
Sun Yat-sen University
Guangzhou, China
dujiangsu@mail.sysu.edu.cn

Hongbin Zhang
Sun Yat-sen University
Guangzhou, China
zhanghb55@mail2.sysu.edu.cn

Taosheng Wei
Sun Yat-sen University
Guangzhou, China
weitsh@mail2.sysu.edu.cn

Zhenyi Zheng
Sun Yat-sen University
Guangzhou, China
zhengzhy37@mail2.sysu.edu.cn

Kaiyi Wu
Sun Yat-sen University
Guangzhou, China
wuky33@mail.sysu.edu.cn

Zhiguang Chen
Sun Yat-sen University
Guangzhou, China
chenzhg29@mail.sysu.edu.cn

Yutong Lu
Sun Yat-sen University
Guangzhou, China
luyutong@mail.sysu.edu.cn

Abstract

Existing LLM serving strategies can be categorized based on whether prefill and decode phases are disaggregated: non-disaggregated (NoDG) or fully disaggregated (FuDG). However, the NoDG strategy leads to strong prefill-decode interference and the FuDG strategy highly relies on high-performance interconnects, making them less cost-effective.

We introduce EcoServe, a system that enables cost-effective LLM serving on clusters with commodity interconnects. EcoServe is built on the partially disaggregated (PaDG) strategy, applying temporal disaggregation and rolling activation for proactive intra- and inter-instance scheduling. It first disaggregates the prefill and decode phases along the time dimension within a single instance to mitigate inter-phase interference and enhance throughput. Next, it coordinates multiple instances and cyclically activates them to ensure the continuous availability of prefill processing, thereby improving latency. Thus, EcoServe's basic serving unit is the macro instance, within which multiple instances collaborate. It further integrates an adaptive scheduling algorithm to route requests in a macro instance and a mitosis scaling approach to enable fine-grained capacity scaling. Beyond delivering high goodput, EcoServe excels in load balancing, hardware cost, parallelism compatibility, and even engineering simplicity compared to existing solutions.

When serving 30B- and 70B-scale models on a production-level cluster with 32 NVIDIA L20 GPUs using commodity Ethernet, EcoServe averagely improves goodput by 82.49%, 86.17%, 122.76%, and 126.96% over four representative NoDG and FuDG systems.

1 Introduction

Large language models [11, 18, 43] (LLMs), have been widely adopted across various tasks [1, 2, 6, 7]. To handle the massive LLM requests, optimizing cost per request while ensuring response times meet service level objectives (SLOs) becomes a primary goal. LLM inference consists of two distinct phases, the prefill phase and the decode phase, each associated with a different SLO, time to first token (TTFT) for the prefill phase and time per output token (TPOT) for the decode phase. The interplay between TTFT, TPOT, and throughput forms an inherent performance trade-off triangle, in which improving one often comes at the cost of the others.

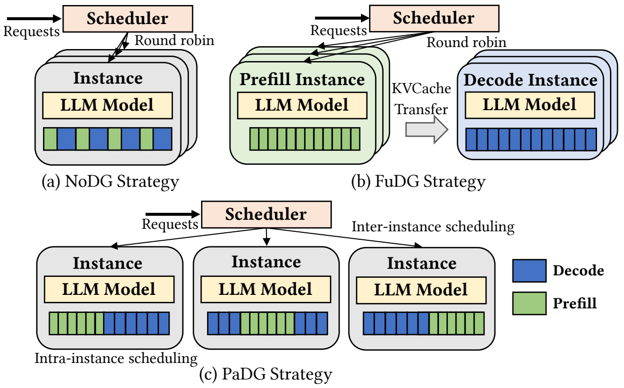


Figure 1. The NoDG, FuDG, and PaDG strategies.

Existing cluster-level LLM serving solutions [3, 5, 9, 33, 35, 48, 50] can be categorized into two strategies based on whether prefill and decode phases are disaggregated: the non-disaggregated (NoDG) strategy and the fully disaggregated (FuDG) strategy. However, both strategies have limitations, either incurring severe interference between prefill and decode phases, or relying heavily on additional hardware capabilities, which prevents them from achieving cost-effectiveness.

Given that the prefill and decode phases share model weights and KV cache, as in Figure 1(a), the NoDG strategy [3, 5, 9, 48], which the prefill phase and the decode phase are placed in a single instance, appears to be a natural choice.

Such colocation inevitably leads to significant interference between the two phases [50]. For example, if prefills are prioritized and inserted excessively for good TTFT, ongoing decodes are poised to experience longer delays, resulting in poor TPOT. Prioritizing the scheduling of one phase risks violating the latency requirements of the other, and this interference also harms throughput, as the decode phase cannot accumulate a sufficiently large batch size to saturate GPU resources. Moreover, the NoDG strategy cannot efficiently adopt pipeline parallelism. Since prefills are varied in lengths and decodes exhibits tight dependency between iterations, micro batch workloads are generally imbalanced and interdependent, resulting in severe pipeline bubbles and further degrading the NoDG strategy's efficiency.

As illustrated in Figure 1(b), the FuDG strategy [33, 35, 50] proposes to fully eliminate the prefill-decode interference by assigning the two phases to separate instances. However, since this strategy requires transferring massive amounts of KV cache between prefill and decode instances, it relies on hyper-clusters with powerful interconnects as the default hardware infrastructure. Unfortunately, high-performance interconnects, such as intra-node NVLINK and inter-node InfiniBand, are not only exceptionally expensive but also power-intensive, even compared to the cost and energy demands of GPUs. Moreover, scaling the performance of FuDG strategy involves both prefill and decode instances, whereas, adjusting the ratio of these two types of instances is challenging, which may lead to significant load imbalance [27]. In addition, on nodes without GPU-direct interconnects, tensor parallelism and KV cache migration intensively contend for PCIe bandwidth, potentially becoming a bottleneck.

In this work, we present EcoServe, an LLM serving system designed to deliver cost-effective LLM inference on clusters with commodity interconnects. As illustrated in Figure 1(c), our key observation is that intra-instance scheduling, which determines when to execute prefills and decodes, must be coordinated with inter-instance scheduling, which decides when and where requests should be routed, to raise the upper bound of the trade-off triangle and fully utilize available resources. To this end, our EcoServe is built on the PaDG strategy, which incorporates temporal disaggregation and rolling activation to achieve proactive intra- and inter-instance scheduling.

The PaDG strategy proactively disaggregates the prefill and decode phases along the time dimension within a single instance. In other words, each instance periodically switches between prefill and decode phases, with each phase lasting longer to reduce switching overhead. Since both phases are still in a single instance, the PaDG strategy avoids KV cache transmission, unlike the FuDG strategy. By mitigating the prefill-decode interference, this approach allows EcoServe to achieve significantly higher throughput.

Next, to meet SLOs, the PaDG strategy further employs a rolling activation scheduling. Without it, if a request is assigned to an instance that has just switched to process the decode phase, it would suffer from an unacceptably high TTFT. Rolling activation proactively coordinates multiple instances in a cyclic pattern. At any given time, there are instances specifically activated for processing prefills, capable of delivering acceptable TTFT latency. A group of such cooperating instances is referred to as a macro instance. Consequently, EcoServe can theoretically satisfy SLOs while also achieving a higher overall throughput, leading to more cost-effective LLM serving. Moreover, since the PaDG strategy minimizes prefill-decode switches and KV cache transmission, it is highly compatible with both tensor parallelism and pipeline parallelism.

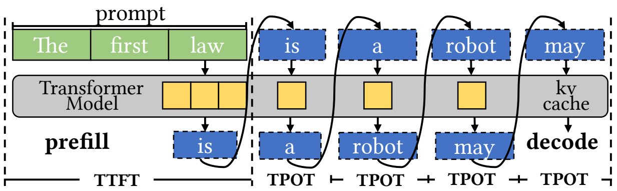


Figure 2. LLM autoregressive decoding process.

With the fundamental concept, EcoServe further includes an adaptive scheduling algorithm and the mitosis scaling approach. The adaptive scheduling algorithm guides request scheduling within the macro instance. While prioritizing the maintenance of satisfactory TPOT, it identifies the most suitable instance for admitting new requests and determines the optimal number of prefill tokens that can be inserted into that instance. The mitosis scaling approach enables elastic and fine-grained control over system capacity by continuously adjusting the number of instances within a macro instance, and triggering a split or merge operation when the instance count exceeds predefined thresholds. To transparently migrate instance between macro instances, it introduces a serializable proxy object that enables logical migration without instance re-initialization and execution interruption . Our contributions are:

- We present EcoServe, a LLM serving system that better enables cost-effective LLM inference on clusters with commodity interconnects.
- We introduce the PaDG strategy, along with the adaptive scheduling algorithm and the mitosis scaling approach.
- We implement EcoServe in a hierarchical architecture.
- We evaluate EcoServe and compare it with four representative serving systems, i.e. vLLM, Sarathi, DistServe, and MoonCake.

2 Preliminary and Motivation

2.1 Computation and Memory of LLM Inferences

As illustrated in Figure 2, the LLM predicts the next token with the accumulated context iteratively until it encounters the end-of-sequence (EoS). By saving the key and value

Table 1. Notations.

Variable	Description	Notation
prompt_len	The length of prompt	S
generation_len	The length of generated tokens	G
batch_size	The number of batched requests	B
layer_num	The number of model layers	L
hidden_size	Input dimension of the hidden layer	H
heads	The number of attention heads	M
size_per_head	The hidden state per head	D

Table 2. Approximate arithmetic intensity (AI) of primary operations in LLMs. As the hidden size is usually large, negligible factors are omitted.

Operation	P/D	FLOPS	Memory Access	Approximate AI
QKV Projection	Prefill	$6BSH^2$	$6BSH + 3H^2$	BS
	Decode	$6BH^2$	$6BH + 3H^2$	B
Attention QK^T	Prefill	$2BS^2H$	$2BSH + BS^2M$	S
	Decode	$2BSH$	$2BSM + BH(S+1)$	1
Attention $(QK^T)V$	Prefill	$2BS^2H$	$2BSH + BS^2M$	S
	Decode	$2BSH$	$2BSM + BH(S+1)$	1
Output Projection	Prefill	$2BSH^2$	$2BSH + H^2$	BS
	Decode	$2BH^2$	$2BH + H^2$	B
Dim Expansion	Prefill	$8BSH^2$	$2BSH + 4H^2$	BS
	Decode	$8BH^2$	$2BH + 4H^2$	B
Dim Reduction	Prefill	$8BSH^2$	$2BSH + 4H^2$	BS
	Decode	$8BH^2$	$2BH + 4H^2$	B

embedding in the memory (i.e. KV cache), redundant computations are avoided in subsequent steps, thus the inference process is divided into prefill and decode phases. Here we conclude their computation and memory features.

$$\mathbf{Q} = W_q\mathbf{X}, \quad \mathbf{K} = W_k\mathbf{X}, \quad \mathbf{V} = W_v\mathbf{X}. \tag{1}$$

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{QK^T}{\sqrt{d_k}})\mathbf{V} \tag{2}$$

$$FFN(x) = Act(xW_1 + b1)W_2 + b_2 \tag{3}$$

Modern LLMs primarily adopt the Transformer architecture [44], which leverages the self-attention mechanism to model complex dependencies in sequences. Its computation involves three main steps:

- **QKV projection** (Equation 1): Each input token is projected into query (**Q**), key (**K**), and value (**V**) embeddings.
- **QKV attention** (Equation 2): A weighed aggregation is performed using the scaled dot-product of queries and keys, determining how each token attends to others in the sequence. The resulting weights are applied to the value vectors, capturing contextual information.
- **Output projection** (Equation 3): A position-wise feed-forward layer further transforms each token's representation with a nonlinear activation, enhancing the model's capacity to learn complex patterns.

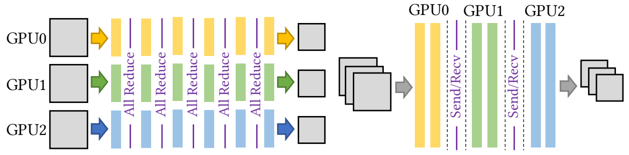
Distinct arithmetic intensity. In Transformer-based LLM inference, matrix multiplications dominate the overall computation time, while softmax and layer normalization account for only a small fraction of the total execution time. As listed in Table 2, there are 6 major matrix multiplication operations and we compute their arithmetic intensities separately for both prefill and decode phases, using the hyperparameters defined in Table 1. The arithmetic intensity is computed by dividing the total number of floating-point operations by the total amount of memory access. Since some terms, such as $1/H$, are negligible, we omit them and present the approximate arithmetic intensity. Although certain optimization techniques, such as FlashAttention [14], can affect the arithmetic intensity, our calculations closely reflect real-world scenarios. As shown in Table 2, the arithmetic intensity of the prefill phase depends on both the sequence length S and batch size B, while the decode phase primarily depends on only the batch size B. Since the sequence length S typically ranges from a few dozen to a few hundred tokens (or even more), the prefill phase exhibits significantly higher intensity compared to the decode phase. Additionally, the decode phase requires loading the KV cache, further increasing memory access. Consequently, when the two phases are executed independently, the prefill phase is generally compute-bound, whereas the decode phase is typically memory-bound.

Memory-compute trade-off. During LLM inference, limited memory capacity can significantly restrict computational parallelism, making parallel inference a potential avenue for achieving superlinear speedup. A key contributor to memory consumption, beyond model weights, is the KV cache. The decode phase is typically bottlenecked by memory bandwidth, necessitating the simultaneous processing of hundreds of requests, which results in substantial memory usage. For example, in Llama-30B, the KV cache for a single token requires 1.52 MB, meaning that 128 requests with an average output length of 300 tokens would demand approximately 58.4 GB of memory, comparable to the memory footprint of model weights. Furthermore, LLM generative tasks inherently involve variable-length sequences, where both input and output lengths are stochastic, and the output length remains unknown until inference completes. This uncertainty necessitates reserving a substantial amount of memory to prevent out-of-memory (OOM) issues, further complicating efficient resource allocation.

2.2 LLM Batching Techniques

To fully utilize modern GPUs, the batching technique is commonly adopted for processing deep learning workloads, where multiple samples are processed simultaneously to expose high parallelism and provide considerable hardware performance improvements. First, given the variation in input and output lengths, continuous batching [48] has emerged as the standard technique, enabling requests to dynamically enter or exit a batch at each iteration. Subsequently, modern LLM serving systems typically employ either separate batching [5] or hybrid batching [9].

Separate batching packs requests exclusively during the prefill phase and solely during the decode phase. Given that



(a) Tensor Parallel (b) Pipeline Parallel

Figure 3. Tensor parallelism and pipeline parallelism.

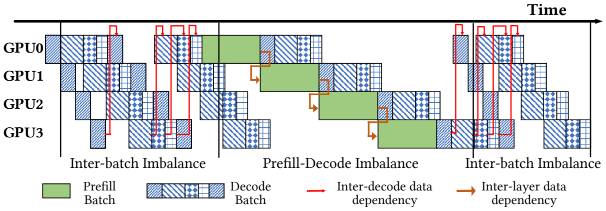


Figure 4. Pipeline bubbles.

these two phases exhibit distinct performance characteristics, i.e. compute-bound and memory-bound, the prefill phase saturates GPU computation even at a batch size of just one, while the decode phase requires a batch size in hundreds. By contrast, hybrid batching combines requests of both prefill and decode phases, and organizes them in a hybrid batch.

2.3 Parallel LLM Inference

To enhance both computational and memory capacity within a single inference instance, various parallelism strategies are employed, including tensor parallelism [13, 38, 40] (TP), expert parallelism [41] (EP), sequence parallelism [22, 29] (SP), and pipeline parallelism [20, 26, 32] (PP). Since TP, EP, and SP all involve distributing the computation of a single layer across multiple devices, they exhibit a similar communication pattern. In this work, we primarily use TP as a representative example to illustrate the key concepts.

Figure 3 illustrates how a model is partitioned in the TP and PP approaches. TP partitions each layer across multiple GPUs, with both model weights and KV cache equally distributed across GPU workers. It demands frequent inter-device communications, with each Transformer layer involves two rounds of all-reduce operations. In our case study of distributed TP inference using Llama-30B on four NVIDIA L20 GPUs (only PCIe), communication overhead accounts for nearly half of the total execution time. While TP can accelerate individual inference runs, it also results in substantial idle time for computational resources due to synchronization and communication bottlenecks.

PP partitions a model layer-wise, where each device is responsible for a subset of layers. It only requires a single point-to-point communication with a much smaller data volume every few layers. The largely-reduced communication makes it a promising approach to support LLM serving, especially on commodity hardware that high-performance interconnects like NVLINK is unavailable. However, the imbalanced pipeline workloads and complex data dependencies of LLM inference often prevent it from being the primary choice. Figure 4 illustrates the execution process of the PP approach with the separate batching, which is commonly adopted in practical LLM serving systems [4, 5].

The imbalanced workloads come from two aspects. First, different batch sizes of decode batches lead to different workloads, resulting in the inter-batch imbalance. Second, the prefill-decode imbalance exists as the execution of prefill batch usually takes much longer time than the decode batch. Likewise, the data dependencies also come from two aspects. Inter-layer data dependency requires data to be processed in the current layer before entering to the next layer for processing. Inter-decode data dependency exists as the execution of generating the next token cannot start until the previous iteration completes. Therefore, bubble problems always hinder the practical use of pipeline parallelism in serving LLMs.

2.4 Large-scale LLM Serving

As demand for LLM inference continues to grow, large-scale deployment and the adoption of cluster-level infrastructure have become essential to meet increasing workload requirements. Based on whether prefill and decode phases are disaggregated, there are NoDG strategy and FuDG strategy.

2.4.1 Non-Disaggregated Strategy.

The NoDG strategy [4, 5, 9] colocates prefills and decodes in a single instance, which is responsible for the entire life-cycle of a request. As shown in Figure 1(a), when a single instance is unable to handle the incoming requests, the system replicates additional instances, each functioning independently to scale out the service. The NoDG strategy supports both separate batching and hybrid batching approaches. As prefills often take much longer than a decoding step, when scheduling together, decodes are always delayed by the prefills, significantly elongating their TPOT; similarly, the inclusion of decodes contributes to a non-trivial increase in TTFT. Thus, NoDG systems often suffer from low throughput as the decode phase can hardly accumulate a sufficiently large batch size to saturate the GPU under SLO constraints. To mitigate this interference, chunked prefill [9] divides a prefill request into smaller chunks and processes a prompt's prefill phase over multiple iterations. However, chunked prefill incurs the overhead of repeated KV cache access, and its effectiveness heavily depends on the input-to-output length ratio.

2.4.2 Fully-Disaggregated Strategy.

As illustrated in Figure 1(b), the FuDG strategy [33, 35, 50] disaggregates the prefill and decode phases across separate instances, with the KV cache transferred between them. When a new request arrives, it first enters a prefill instance, where the KV cache and the first token are generated. The KV cache is then transmitted to a decode instance for the remaining decoding steps.

Table 3. KV cache generation speed in the prefill instance and theoretical bandwidth required for the FuDG strategy. Here each node includes 8 GPUs and tensor parallelism is applied when a single GPU's memory capacity is insufficient.

Model	Device	Tokens/s	Theoretical Bandwidth
Llama-30B	L20	6584.6	9.796 GB/s
Llama-30B	A800	26189.2	38.96 GB/s
CodeLlama-34B	L20	6838.92	1.25 GB/s
CodeLlama-34B	A800	25978.88	4.76 GB/s

Although it can completely eliminate the prefill-decode interference, it introduces issues related to data transmission and load imbalance [27].

Since the FuDG strategy requires transferring massive amounts of KV cache data between prefill and decode instances, it relies on high-performance interconnects to avoid bottlenecks. Table 3 presents the KV cache generation speed in a GPU node (all prefill instances), along with the the theoretical bandwidth required to transfer these KV cache data. When deploying Llama-30B prefill instances on a node equipped with 8 NVIDIA A800 GPUs, the theoretical bandwidth required to transfer the generated KV cache data off the node exceeds 38 GB/s, necessitating at least a 400 Gbps network to sustain the throughput. Although Grouped Query Attention (GQA) [10] in CodeLlama-34B significantly compresses KV cache size, the strategy still demands over 4.76GB/s bandwidth, requring a 50Gbps network.

Consequently, to support deployment across clusters with varying interconnect capabilities, the FuDG strategy can be further classified into intra-node FuDG (DistServe [50]) and inter-node FuDG (MoonCake [35]), depending on whether the prefill and decode instances are colocated within the same node or distributed across different nodes. In scenarios where inter-node interconnects are insufficient, DistServe deploys prefill and decode instances within a node and mitigates the issue by transferring KV cache data over intra-node high-speed links, such as NVLink, although these interconnects are also costly. In contrast, MoonCake designs a centralized KV cache pool and relies on InfiniBand to connect prefill and decode instances across nodes. In short, the FuDG strategy incurs substantial high-performance networking requirements in both intra-node and inter-node setups.

Next, the FuDG strategy also suffers from severe load imbalance issues. Firstly, it requires careful load balancing between prefill instances and decode instances. Due to the asymmetry in durations, adding a single prefill instance typically necessitates provisioning multiple decode instances to maintain load balance. If prefill and decode instances are colocated in a single node due to bandwidth limitations, achieving load balance becomes somewhat infeasible. Secondly, memory utilization across prefill and decode instances is imbalanced. Decode instances store large amounts of KV cache, while prefill instances store much less. As memory capacity is a valuable resource in modern LLM workloads, such imbalance may leave a significant portion of memory idle in prefill instances, resulting in suboptimal resource efficiency.

3 EcoServe Design

3.1 Overview

As depicted in Figure 5, EcoServe employs a partially disaggregated (PaDG) strategy that proactively orchestrates both intra- and inter-instance execution to optimize goodput. Specifically, the prefill and decode phases are disaggregated along the time dimension in each instance (❶ Temporal Disaggregation), while coordination is conducted across different instances to ensure continuous availability (❷ Rolling Activation).

The EcoServe system is organized in a hierarchical architecture, comprising three levels of scheduling: the overall scheduler, the macro-instance scheduler, and the instance scheduler. The instance scheduler (❺) is responsible for managing execution within a single instance, including coordinating prefill and decode phases, orchestrating multiple devices, and executing the directives from higher-level schedulers. The macro-instance scheduler coordinates multiple instances by aggregating their execution states and dispatching requests to appropriate instance according to given profiling results and service-level objectives (SLOs), where the macro instance (❻) is defined as a unique abstraction level introduced in EcoServe and serves as the smallest scheduling unit in the system. The overall scheduler (❼) dispatches requests to macro-instances based on their capabilities. Besides, it manages the capacity scaling in different macro instances, such as transferring instance handler between macro instances. In this work, we primarily focus on the internal architecture within a macro-instance.

Furthermore, EcoServe integrates the adaptive scheduling algorithm and the mitosis scaling approach to enable practical implementation of the PaDG strategy. The adaptive scheduling algorithm (❸) is applied at both the instance scheduler and macro-instance scheduler and makes them coordinated. From the perspective of the instance scheduler, it executes decodes while accumulating sufficient slack to safely admit new requests, until it receives a continuous stream of incoming requests from the macro-instance scheduler. From the perspective of the macro-instance scheduler, it continuously receives new requests from the overall scheduler and updates execution status from individual instances. Based on this information, it estimated which instance and how many prefill tokens should be forwarded under given constraints.

The mitosis scaling approach (❹) enables fine-grained capacity adjustments to accommodate fluctuations in LLM inference workloads over extended periods of time. Since

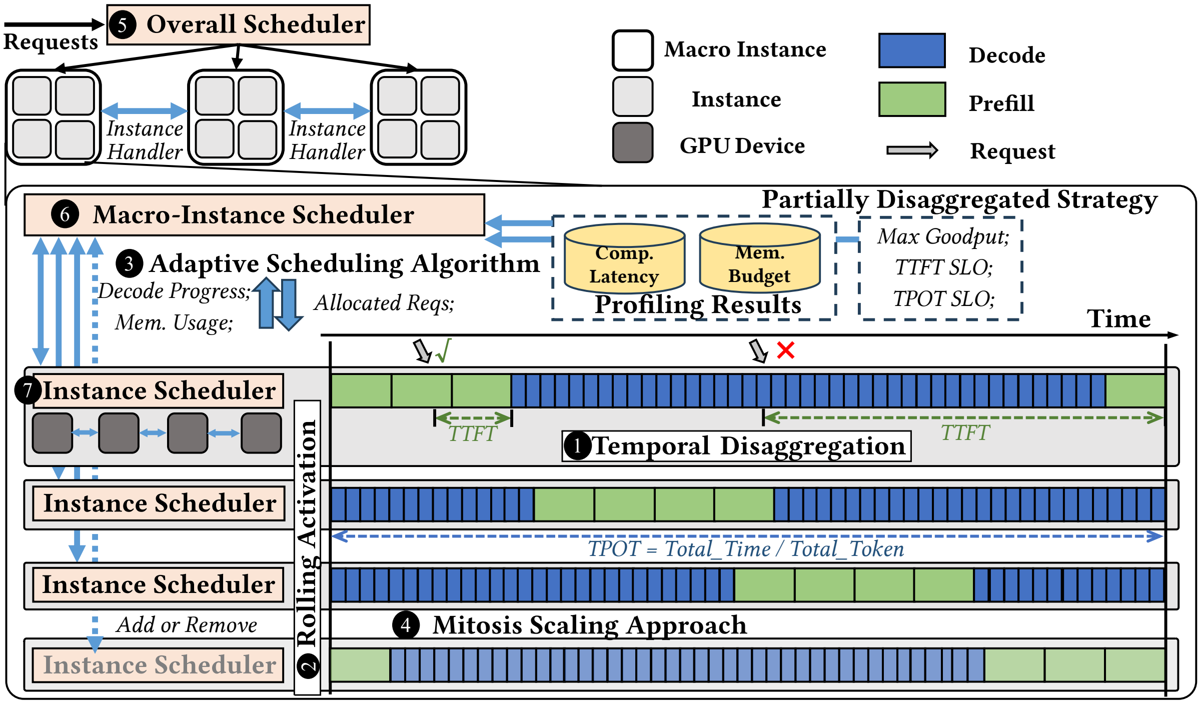


Figure 5. EcoServe Architecture Overview.

the smallest scheduling unit in EcoServe is the macro instance, adjusting capacity at the granularity of an entire macro instance can be inflexible and often leads to resource under utilization or waste. Inspired by biological cell mitosis, this scaling strategy incrementally adds or removes instances within a macro instance in response to changed demand. Once predefined thresholds are reached, it then splits a macro instance or merges two macro instances, enabling elastic adaptation to workload changes with fine-grained control. Further, to avoid instance re-initialization and execution interruption, a serializable proxy object is designed for flexible instance migration between macro instances.

3.2 Partially Disaggregated Strategy

Figure 5 contains the proactive intra-instance and inter-instance scheduling of the PaDG strategy. The proactive intra-instance scheduling, referred to as temporal disaggregation, reduces prefill-decode interference and enhance throughput, though at the cost of increased TTFT. The proactive inter-instance scheduling, guided by rolling activation, plays a key role in rescuing TTFT, ensuring timely processing of new requests.

3.2.1 Temporal Disaggregation. To mitigate prefill-decode interference, the PaDG strategy disaggregates the prefill and decode phases along the temporal dimension within each instance. Unlike the FuDG strategy, which assigns distinct prefill and decode roles to different instances, PaDG assigns these roles to different time slots within the same instance. This design preserves execution locality while mitigating prefill-decode interference, thereby achieving efficient on-device resource utilization and eliminating cross-instance data transmission.

Through proactive intra-instance scheduling, each inference instance processes only one phase type at a time for an extended duration. As illustrated in Figure 5, when a new request arrives at an instance that is currently processing prefills, it can be immediately processed, thereby meeting the TTFT SLO. However, if the target instance is currently performing the decode phase, the request must wait until the instance transitions to the prefill phase, leading to an unacceptable increase in TTFT. Consequently, since each phase occupies an instance for an extended period, this intra-instance scheduling significantly degrades TTFT, making it difficult to meet the corresponding TTFT SLO. In comparison, modern LLM serving systems typically render outputs in a typewriter mode, where the TPOT SLO can be satisfied as long as a sufficient number of tokens are generated within a time window. This means that if the decode execution is faster than the TPOT constraint, it can accumulate spare time (referred to as saved TPOT), which can be used for interruptions in the decode phase without violating the SLO.

3.2.2 Rolling Activation. Although a single instance can only remain in either the prefill or decode phase at any given time, and therefore cannot immediately process newly arrived requests, rolling activation proactively schedules multiple instances and staggers their prefill phases over time,

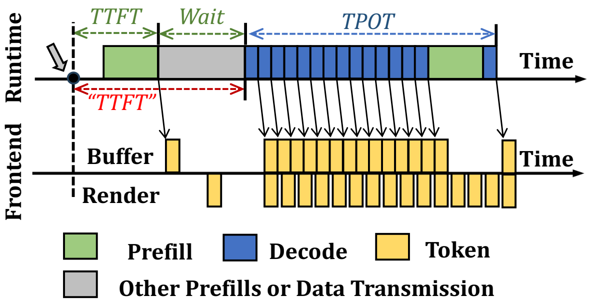


Figure 6. Runtime and frontend Timing.

thereby ensuring the continuous availability of prefill processing. As illustrated in Figure 5, these instances are activated to perform prefill phases in a cyclic pattern. From the perspective of individual requests, they are always routed to instances that are currently in the prefill phase and can be processed intermediately. In this way, rolling activation reduces waiting time and rescues TTFT. Since the output length is undetermined, instances require to constantly update their statuses, such as decode progress and memory usage, to macro instance for coordination.

3.3 System Metrics v.s. User Experience

Figure 6 demonstrates the timing characteristics at both the runtime and frontend. Once a token is generated, it is transmitted to the frontend, where it is buffered before being rendered. From the runtime perspective, classical metrics such as Time to First Token (TTFT) and Time Per Output Token (TPOT) are commonly used to characterize the latency behavior of a request's prefill and decode phases, respectively. However, for existing LLM serving systems, they have been insufficient to reflect the service quality.

TTFT and TPOT alone are insufficient to capture the performance characteristics of the prefill-decode switching phase. Before a request enters its decode phase, additional operations occur across all strategies, NoDG, PaDG, and FuDG. For the NoDG and PaDG strategies, prefills from other requests may occur before a given request can enter its decode phase. Similarly, the FuDG strategy incurs KV cache transmission overhead prior to the decode phase. Thus, the phase-switching waiting time should be introduced to provide a more comprehensive evaluation of LLM serving systems. This metric has implicitly appeared in previous work, and it is frequently misrepresented, potentially masking key limitations in existing LLM serving systems.

To maintain consistency with previous studies, we continue to use TTFT as the metric for evaluating prefill latency. However, in this context, the reported TTFT actually encompasses two components: the true TTFT and the phase-switching waiting time. Notably, this definition of TTFT represents **a stricter SLO**, as it includes additional overhead within the same latency constraint. Accordingly, the measurement of TPOT begins after the phase-switching delay.

Algorithm 1: Inter-Instance Scheduling Algorithm

```
Data: current request: $req$; instance list: $instances$;
Function InterSchedule($req$):
    prev_idx ← last request's routed instance;
    last_instance ← $instances[prev\_idx]$;
    if CheckConstraints ($last\_instance$,req) then
        route $req$ to $instance[prev\_idx]$;
    else
        next_idx ← $(prev\_idx + 1)\%len(instances)$ ;
        route $req$ to $instance[next\_idx]$;
```

Algorithm 2: Constraint Checking Algorithm

```
Data: System constraints: $SLO_{TTFT}$, $SLO_{TPOT}$;
Function CheckConstraints($instance, req$):
    Constraint 1: TTFT
    $t_{\text{switch}}$ ← phase switching timestamp;
    $pending\_prefills \leftarrow \{r \in instance.reqs \mid r.arrival\_time \geq t_{\text{switch}}\} \cup \{req\}$
    $prefill\_times$ ← predict $pending\_prefills$ durations ;
    $t_{\text{total}} \leftarrow \sum prefill\_times$;
    if $t_{\text{total}} > \text{SLO}_{\text{TTFT}}$ then
        return NotSatisfied;
    Constraint 2: TPOT
    $existed\_decodes \leftarrow \{r \in instance.reqs \mid r.arrival\_time < t_{\text{switch}}\}$;
    $saved\_tpots$ ← [];
    $current\_time$ ← current timestamp;
    foreach $r \in existed\_decodes$ do
        $L \leftarrow r.output\_length$;
        $saved\_tpot \leftarrow L \times SLO_{TPOT} - (current\_time - r.first\_token\_time)$
        $saved\_tpots$.append($saved\_tpot$)
    $mean\_saved\_tpot \leftarrow mean(saved\_tpots)$;
    if $mean\_saved\_tpot < t_{\text{total}}$ then
        return NotSatisfied;
    Constraint 3: KV Cache capacity
    if $req\_kvcache\_size > remain\_memsize$ then
        return NotSatisfied;
    return Satisfied
```

3.4 Adaptive Scheduling Algorithm

To enable proactive scheduling within and across instances, we propose an adaptive scheduling algorithm consisting of three sub-algorithms: the Inter-Instance Scheduling Algorithm, the Intra-Instance Scheduling Algorithm, and the Constraint Checking Algorithm, coordinating decisions at multiple levels.

The inter-instance and the intra-instance scheduling algorithms guide the fundamental execution of the instance macro-scheduler and the instance scheduler. In general, the macro-instance scheduler and the instance scheduler function in a master-slave manner. From the instance scheduler's perspective, although the final outcome is that prefill and decode phases are disaggregated along the temporal dimension, its scheduling algorithm prioritizes prefills. It continues processing active decodes, periodically updating its progress to the macro-instance scheduler, and switches to prefills upon receiving new requests from the macro-instance scheduler.

From the macro-instance scheduler's perspective, it receives status updates from all instances and schedules them to achieve rolling activation. Specifically, the inter-instance scheduling algorithm traverses all instances and routes requests cyclically. As shown in Algorithm 1, for an incoming request, the algorithm first attempts to route it to the same instance that processed the previous request. If the selected instance meets the constraints verified by the Constraint Checking Algorithm, the request is forwarded to this instance. If the instance cannot satisfy the constraints, the algorithm will check the next available instance.

The constraint checking algorithm, as described in Algorithm 2, is responsible for verifying that assigning an incoming request to an instance will not violate the TTFT/TPOT latency SLOs or exceed the instance's available memory capacity. First, the algorithm ensures that the total duration (denoted as t_{total}) will not exceed the SLO_{TTFT} after adding a new request to the pending prefills during this prefill phase. The prefill duration of a single request can be predicted in advance by profiling sequences of various lengths. This ensures the TTFT constraint, as outlined in Section 3.3. Additionally, by utilizing updated information from the instance, the algorithm calculates the saved TPOT by subtracting the required time from the achieved time. As discussed in 3.2.1, provided that t_{total} does not exceed the saved TPOT, the TPOT constraint will be satisfied. Finally, the algorithm ensures that the combined KV cache size of the requests does not exceed the remaining GPU memory capacity, preventing memory overflow during processing.

3.5 Mitosis Scaling Approach

Although the macro instance is the smallest scheduling unit in EcoServe, scaling capacity at this granularity is often inefficient and inflexible. To address this limitation, the mitosis scaling approach, inspired by biological cell mitosis, provides a solution to adjust capacity at the instance level, allowing EcoServe to adapt more precisely to workload fluctuations. In general, this strategy first adds or removes instances within a macro instance, and subsequently adds or removes entire macro instances through splitting or merging.

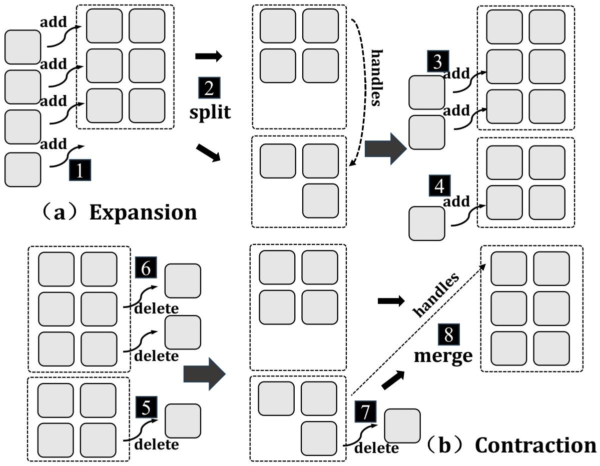


Figure 7. The illustration of the expansion and contraction processes. Here $N_l = 3$ and $N_u = 6$.

3.5.1 Expansion and Contraction.

We initially set two hyperparameters, N_l and N_u, representing the lower and upper bounds on the number of instances in a macro instance. If all instances are symmetry in network topology, the number of instances within a macro instance can theoretically range from 1 to infinity. However, a small N_l may lead to frequent phase switching, while a large N_u can introduce instance management overhead and potentially become a scheduling bottleneck.

Figure 7 illustrates an example of how the expansion and contraction processes are performed within the system. Here the scaling can be triggered either when the system fails to meet the defined SLOs or when there is sustained resource underutilization. New instances are incrementally added until the number of instances exceeds the upper limit N_u, at which point a new macro instance containing N_l instances is split off from the original macro instance (step 2). If additional instances are still required, they are first added to the original macro instance until it again reaches N_u (step 3), and subsequent instances are then added to the new macro instance (step 4).

On the contrary, when the capacity becomes excessive and the contraction process is triggered, instances are firstly removed from the smallest macro instance until the number of instances in the macro instance reaches N_l (step 5). Next, instances start to be removed from a full macro instance (step 6). When the total number of instances across these two macro instances reaches N_u (step 7), they will be merged into a single macro instance after one additional instance is removed (step 8). After expansion or contraction, each macro instance continues scheduling according to the adaptive algorithm, with no additional logic required. Thus, the

Table 4. Dataset Features and Corresponding SLOs.

DataSet	In_{Avg}	In_{Med}	Out_{Avg}	Out_{Med}	SLO_{TTFT}	SLO_{TPOT}
Alpaca-gpt4	20.63	17.00	163.80	119.00	1s	100ms
ShareGPT	343.76	148.00	237.20	152	5s	100ms
LongBench	2686.89	2736.50	101.78	19	15s	100ms

system generally maintains multiple full macro instances, along with one or two partially filled macro instances.

3.5.2 Flexible Instance Migration. To dynamically split or merge macro instances without reinitializing or interrupting individual instances, we design a serializable proxy object that enables instance handles to be transferred between different macro-instance schedulers. At the core of this design is the *InstanceHandler* metadata, which encapsulates essential information such as the actor ID, worker address, function calls, and other relevant attributes. When a handler is transferred between macro-instance schedulers (i.e., across processes), it is first serialized using the pickle library and then sent to the target macro-instance scheduler. The transmission process is coordinated by the overall scheduler. Upon deserialization, the receiving process reconstructs a fully functional proxy, which can issue function calls through the RPC-like system. This design enables logical migration of instances across macro-instance schedulers without interrupting their execution, thereby supporting more flexible and low-overhead scaling.

4 Evaluation

EcoServe utilizes vLLM as the single-device runtime, with Ray controlling multiple devices per instance through RPC-like control, while ZeroMQ facilitates synchronization across instances in the macro-instance scheduler. We evaluate EcoServe across LLMs of varying sizes and on diverse application datasets and clusters.

4.1 Experimental Setup

Cluster testbed. We conduct our experiments on two clusters. The primary testbed is a production-level cluster deployed within a technology company, representing a typical infrastructure setting in modern data centers. This cluster consists of 8 nodes with a total of 64 GPUs, each node equipped with 8 NVIDIA L20-48GB GPUs connected via PCIe only. These nodes are interconnected through a standard 10Gbps Ethernet. The second testbed consists of two nodes, each equipped with 8 NVIDIA A800-80GB GPUs, with all GPUs connected via PCIe only. Unlike the primary cluster, this setup features a higher-bandwidth interconnect between nodes, i.e. 25Gbps RoCE.

Model, dataset and workloads setup. We choose three representative LLM models, i.e. Llama-30B [43], CodeLlama2-34B [36], and Qwen2-72B [46], in our experiments. LLaMA-30B adopts the standard multi-head attention (MHA) mechanism, while CodeLlama2-34B and Qwen2-72B employ the emerging grouped-query attention (GQA) mechanism [10]. By sharing keys and values across multiple query heads, GQA significantly reduces KV cache size during inference, thereby alleviating the transmission overhead associated with the FuDG strategy. We use BF16 precision in all experiments.

For target applications and corresponding datasets, as in Table 4, we select three representative applications with diverse input and output length distributions and remove outlier samples by truncating inputs to a maximum length of 4096, following prior studies [24, 35, 50].

- **Alpaca-gpt4:** This dataset is used for the human instruction application. As shown in Table 4, it is characterized by short input sequences and relatively long outputs, with the average output length approximatebly 10 times longer than the input length.
- **ShareGPT:** The dataset refers to the chatbot application, featuring relatively balanced input and output lengths.
- **LongBench:** This dataset is used for the summarization application, where the goal is to generate a concise summary for a long article. As a result, it is characterized by long input sequences and short outputs.

We set TTFT and TPOT SLOs based solely on applications, without differentiating between model sizes. Our SLOs are, in most cases, stricter than those in prior works [35, 50]. We pair each model with each dataset to construct multiple alternative workloads. To emulate realistic serving, a Poisson distribution is applied to a fixed request rate to introduce minor fluctuations.

Baseline. We compare EcoServe against four baseline systems in NoDG or FuDG strategies. We follow their released implementations, and luckily, all baselines are built with vLLM [5] as the underlying runtime, ensuring fairness.

- **vLLM [5]:** This is the NoDG strategy with separate batching and prefill-priority scheduling, which is originally provided by vLLM system.
- **Sarathi [9]:** This is the NoDG strategy with hybrid batching, decode-priority scheduling, and the chunked prefill technique.
- **DistServe [50]:** The intra-node FuDG strategy that prefill and decode instances colocate in the single node. Notably, while DistServe includes a strategy that distributes each instance across nodes and limits the KV cache transmission within a node, this strategy is only applied to pipeline parallelism and cannot satisfy SLOs in our setting.
- **MoonCake [35]:** The inter-node FuDG strategy that prefill and decode instances can be assigned to different nodes. MoonCake introduces an intermediate KV cache pool, which acts as a centralized buffer for KV cache transmission. Even when the prefill and decode instances reside on the same node, KV cache still needs to be transferred through this intermediate pool. To mitigate the load imbalance issue, we perform different P/D ratio and select the optimal one.

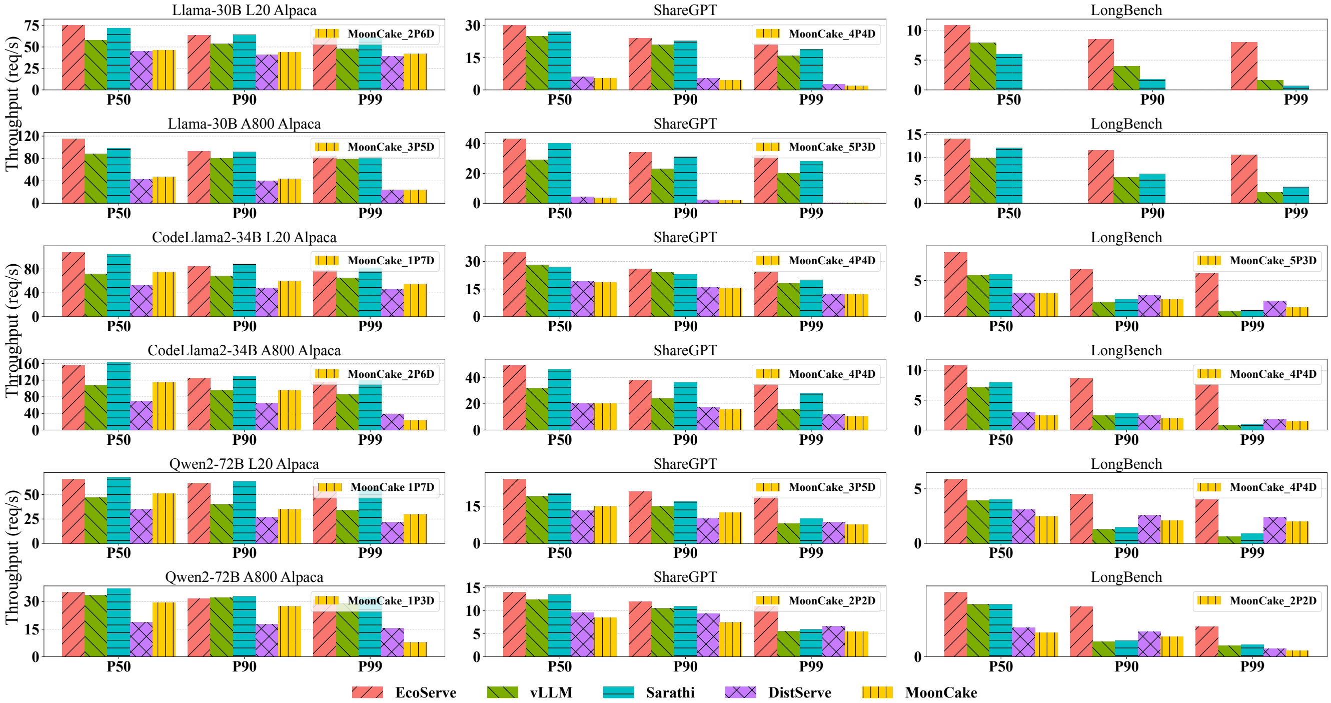


Figure 8. End-to-end performance comparison. MoonCake and DistServe cannot meet SLOs in Llama-30B with LongBench.

Metrics. We also use SLO attainment as the evaluation metric following prior works. Specifically, we compare system throughput under different levels of SLO attainment, including the P50, P90, and P99 percentiles. The throughput is collected by incrementally increasing the request rate until the system fails to reach the attainment.

4.2 End-to-end Performance Evaluation

We compare EcoServe against baselines across full combinations. For the L20 cluster, we employ 32 GPUs in 8 nodes and configure the models with tensor parallelism (TP): Llama-30B and CodeLlama2-34B are both run with TP=4, while Qwen2-72B is configured with TP=8. To alleviate the bandwidth limitations in MoonCake, we deploy a single instance per node, reducing the bandwidth contention, and there are 8 instances for each model. For the A800 cluster, we use 16 GPUs and configure LLaMA-30B and CodeLlama2-34B with TP=2, and Qwen2-72B with TP=4. Accordingly, LLaMA-30B and CodeLlama2-34B each have 8 instances, while Qwen2-72B has 4 instances.

Overall Comparison. In Figure 8, EcoServe outperforms baselines in most cases. For NoDG systems, EcoServe achieving an average P90 goodput improvement of 83.76% over vLLM and 71.97% over Sarathi. By mitigating the prefill-decode interference, the PaDG strategy provides a larger room for balancing TTFT and TPOT through cross-instance cooperation. NoDG systems can still achieve comparable or even slightly better performance than EcoServe, such as when serving the Alpaca dataset. Since the Alpaca dataset features very short input lengths, and SLOs are already loose enough that the extra trade-off space offered by PaDG becomes less impactful. For FuDG systems, although FuDG systems can deliver performance comparable to or better than NoDG systems when serving models with reduced KV cache and datasets with relatively long outputs, they fall significantly behind EcoServe. EcoServe achieves an average P90 goodput improvement of 192.41% over DistServe and 218.22% over MoonCake.

Comparison Across SLO Attainment Levels. All systems experience a decline in throughput as the SLO attainment level increases from P50 to P99. However, EcoServe demonstrates higher tolerance in tighter SLOs. At P50 SLO attainment, EcoServe achieves 36.49%, 19.82%, 180.73%, and 194.62% higher throughput compared to the baselines. Under the tighter P90 SLO attainment, these improvements increase significantly to 83.76%, 71.97%, 192.41%, and 218.22%. Next, this gap further widens, and some baseline systems are unable to meet the P99 SLO attainment. This validates that PaDG can provide a larger room for balancing TTFT and TPOT through inter-instance cooperation.

Comparison Across Models. EcoServe averagely outperforms NoDG systems' throughput by 65.00%, 83.30%, and 85.30% for serving Llama-30B, CodeLlama2-34B, and Qwen2-72B models under P90 SLO attainment, demonstrating consistent performance improvements across models. In contrast, when compared to FuDG systems, EcoServe's advantage under P90 SLO attainment varies significantly across models, achieving 507.67%, 125.45%, and 83.61% throughput improvements respectively. Llama-30B presents significant performance degradation in FuDG systems, primarily due to its larger KV cache size, while CodeLlama2-34B and Qwen2-72B use the emerging GQA [10] with reduced size, thereby alleviating transmission overhead. Compared to CodeLlama2-34B, Qwen2-72B performs better. Since the computational cost grows quadratically with model size, Qwen2-72B has relatively smaller KV cache.

Comparison Across Clusters. Under P90 SLO attainment, EcoServe achieves an average throughput improvement of 71.41% over NoDG systems and 285.78% over FuDG systems on the A800 cluster, and 84.33% over NoDG systems and 124.86% over FuDG systems on the L20 cluster. The A800 and L20 clusters exhibit a similar performance trend when compared to NoDG systems, but show notable differences in their comparison with FuDG systems. Although the A800 cluster is equipped with higher bandwidth, it appears less favorable for FuDG systems. As shown in Table 3, while the bandwidth increases by 2.5×, the processing capability improves by over 4×, thereby making the inter-node network an even more significant bottleneck.

Comparison Across Applications. Under P90 SLO attainment, EcoServe achieves an average throughput improvements of 10.44%, 20.60%, and 202.57% over NoDG on the Alpaca, ShareGPT, and LongBench datasets, while outperforming FuDG by 74.80%, 363.10%, and 164.42% on these datasets. The improvement over FuDG on the LongBench would be higher, as we exclude its performance on Llama-30B due to execution failures. Comparing with NoDG systems, shorter input lengths result in reduced prefill-decode interference and fewer repeated accesses to KV cache during chunked prefill, allowing EcoServe to perform better. In the case of FuDG, datasets with longer input and relatively shorter outputs require more prefill instances to generate KV cache, increasing network transmission pressure and leading to worse performance.

4.3 Scaling Capability

This section evaluates the scaling efficiency of EcoServe. First, we double the number of instances and assess its goodput capability. Next, we evaluate the fine-grained scaling ability under dynamically incremental request rate.

4.3.1 Static Coarse-grained Scaling. This section evaluates the static scaling as the available resources are increased by a factor of two. The CodeLlama2-34B and Qwen2-72B models are tested on the L20 cluster, using TP=4 for CodeLlama2-34B and TP=2 for Qwen2-72B, with the ShareGPT dataset as the workload.

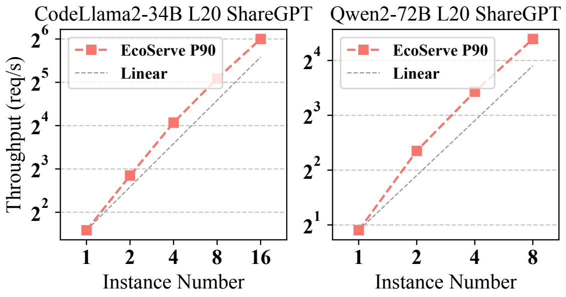


Figure 9. Static coarse-grained scaling.

Figure 9 demonstrates that both models' serving achieve superlinear improvement under P90 SLO attainment. For instance, when scaling from 1 instances (4 GPUs) to 4 instances (16 GPUs), the CodeLlama2-34B serving achieves 5.6× throughput under P90 SLO attainment. First, this can be attributed to the fact that EcoServe incurs minimal overhead in managing more instances within a macro instance and the nodes in this cluster are symmetrical in topology. More importantly, adding more instances lead to more space for mitigating inter-phase interference, enabling higher arithmetic intensity and better GPU saturation. Assuming a macro instance contains only a single instance, the PaDG strategy actually degrades to the NoDG strategy, and two phases still switch frequently and interference severely in a single instance. This superlinear scaling effect will plateau once a sufficient number of instances is reached.

4.3.2 Dynamic Fine-grained Scaling. This section evaluates the fine-grained scaling, where individual instances are incrementally added to a macro instance as request rates increase. We use CodeLlama2-34B on the L20 cluster with TP=4, and the ShareGPT dataset as the workload. The request rate is gradually increased every 2 minutes, ranging from 20 to 50 requests per second, and SLO attainments are collected every 30 seconds. Based on the weak scaling experiments, we set the hyperparameters to $N_l = 4$ and $N_u = 16$, as division of the macro instance would lead to performance degradation. The system starts with 8 instances and finally uses up all GPUs. As shown in Figure 10, increasing the request rate initially results in a drop in SLO attainment, which is then restored by the addition of a new instance. The adaptive scheduling algorithm can immediately route new requests to the newly added instance, leaving more time slots for existing instances to process decodes.

Although the node number does not necessarily trigger macro instance splitting and instance migration, we further assess the serializable proxy object in the mitosis scaling approach when serving at larger scale. The use of the serializable proxy object ensures that the migration process does not interrupt instance execution, introducing less than 100 ms of overhead. This overhead can be entirely hidden by triggering the migration during the decode phase. In contrast, interrupting and re-initializing an instance incurs much higher

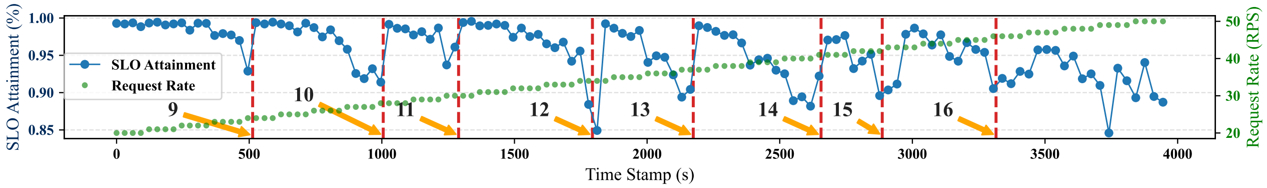


Figure 10. Dynamic fine-grained scaling. Individual instances are dynamically added to a macro instance as request rates increase. Here $N_l = 4$ and $N_u = 16$.

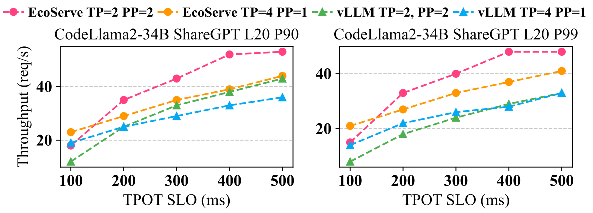


Figure 11. Pipeline parallel compatibility.

overhead. For instance, re-initializing CodeLlama2-34B from L20 node's local storage takes about 3 minutes, and this can be further prolonged when weights are loaded from remote storage. In conclusion, the mitosis scaling approach can provide flexible and fine-grained scaling, effectively adapting to dynamic workload demands.

4.4 Parallelism Compatibility

We further validate that the FuDG strategy is better suited for pipeline parallelism (PP) compared to the NoDG strategy. We use CodeLlama2-34B, ShareGPT, and the L20 cluster. As PP does not improve the latency of a single batch, a relaxed TPOT constraint is required. Figure 11 illustrates the throughput as the TPOT SLO increasing from 100ms to 500ms. In this setup, CodeLlama2-34B is configured with TP=2, PP=2 and TP=4, PP=1. It is evident that EcoServe, when utilizing PP, achieves better performance than its TP counterpart at lower TPOT SLOs, outperforming vLLM. In other words, the intersection point occurs at a slower TPOT SLO and the throughput plateau achieved with PP is much higher than that of vLLM.

5 Related Work

Scheduling in LLM serving. Based on whether prefill and decode phases are disaggregated, existing LLM serving approaches can be categorized into the NoDG strategy [4, 5, 9, 48] and the FuDG strategy [33, 35, 50], which are most relevant to EcoServe. Adrenaline [27] notices the load imbalance issue in FuDG and reschedules computation in prefill and decode instances.

Moreover, other studies address issues in specific inference scenarios. Flexgen [39], FastDecode [19] and Specinfer [31] enable LLM inference with limited memory capacity by employing offloading strategies. Loongserve [45] and Infinite-llm [28] targets long-context inference and optimize parallel strategy and memory utilization respectively. Moe-lightning [12], Pre-gated MoE [21] and Lina [25] focus on MoE models and optimize resource utilization by employing expert popularity. MegaScale-Infer [52] targets ultra-large MoE model and it accelerates the decode phase by disaggregating the attention and FFN modules. Liger [15] and NanoFlow [51] carefully schedules and overlaps GPU kernels from different requests to improve efficiency.

KV Cache Management. To reduce KV cache memory usage of standard MHA [44], MQA [37] and GQA [10] share key and value projections across query heads. PagedAttention [24] and vAttention [34] reduces memory fragmentation by organizing the KV cache into fixed-size blocks. To compress KV cache, H2O [49], Keyformer [8], and Liu et al. [30] find token similarity and removes redundant information. Next, Shadowkv [42], Prompt cache [17], and Ragcache[23] further explore KV cache compression and offloading strategies in long-context scenarios. AttentionStore [16] schedules KV cache across hierarchical storage tiers, while CacheBlend [47] introduces the pipelining loading with partial recomputation to use slower object stores.

Table 5. Large-scale LLM Serving Strategy Comparison.

	Goodput	Cost Effective	Load Balance	Hardware Cost	Parallelism Compatibility	Engineering Complexity
NoDG	✓	Good	Easy	Low	Low	Low
FuDG	✓✓	Poor	Hard	High	High	High
PaDG	✓✓	Excellent	Easy	Low	High	Low

6 Discussion

Commercial success in LLM serving hinges on adopting cost-effective strategies on large-scale clusters, which **require a careful trade-off between throughput, SLO attainment, infrastructure cost, parallelism compatibility, and even engineering complexity**. Table 5 presents a comparison between the PaDG strategy and the existing NoDG and FuDG strategies. In terms of goodput, the PaDG strategy is comparable to FuDG, while largely outperforming NoDG. While FuDG is designed for tight SLOs and relies on high-performance interconnects, PaDG is optimized for cost-effective deployments. When SLOs are relaxed or interconnects are limited, PaDG can outperform FuDG.

Beyond hardware, PaDG also reduces load imbalance and engineering complexity. Unlike FuDG, which scales across two instance types, both NoDG and PaDG scale at the granularity of individual instances, leading to simpler scaling and more balanced workloads. In addition, the lack of cross-instance KV cache transmission in PaDG and NoDG significantly lowers system complexity. From a parallelism compatibility perspective, PaDG offers further advantages: its lower frequency of prefill-decode switching improves pipeline parallelism efficiency, while minimal data movement and reduced PCIe contention make it more suitable for tensor parallelism on systems without direct GPU interconnects.

NoDG, PaDG, and FuDG each have their own advantageous scenarios, and LLM serving vividly demonstrates the art of trade-offs in system optimization. NoDG is well-suited for small models, such as 7B and 13B. These models have lower computational demands, and their SLOs are easier to satisfy, making prefill-decode interference negligible. Larger models, such as 30B, 70B, and 130B, benefit more from PaDG. These models typically require parallel techniques to extend memory capacity and are still capable of meeting typical latency SLOs in a single instance. In extreme scenarios, like ultra-large models or stringent SLOs, even minor interferences can significantly degrade these metrics, FuDG with advanced hardware becomes essential. More aggressive strategies are also worth studying. For example, MegaScale-Infer [52] studies ultra-large MoE model and disaggregates the attention and FFN modules into different instances. Moreover, these strategies incur incremental engineering costs, which also serve as a major factor.

7 Conclusion

This paper presents EcoServe, a cost-effective LLM serving system with a novel Partially Disaggregated (PaDG) Strategy. The PaDG strategy leverages proactive intra and inter-instance scheduling to better balance TTFT and TPOT, significantly improving throughput on clusters with commodity interconnects.

References

[1] 2023. Github Copilot. https://github.com/features/copilot

[2] 2024. Chatgpt. https://chat.openai.com

[3] 2024. Faster Transformer. https://github.com/NVIDIA/FasterTransformer

[4] 2024. SGLang. https://github.com/sgl-project/sglang

[5] 2024. vllm: Easy, fast, and cheap llm serving for everyone. https://github.com/vllm-project/vllm

[6] 2025. Character ai. https://character.ai

[7] 2025. Cursor. https://www.cursor.com

[8] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. Proceedings of Machine Learning and Systems 6 (2024), 114–127.

[9] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} Tradeoff in {LLM} Inference with {Sarathi-Serve}. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). 117–134.

[10] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv:2305.13245 [cs.CL] https://arxiv.org/abs/2305.13245

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. Advances in neural information processing systems 33 (2020), 1877–1901.

[12] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E Gonzalez, Matei Zaharia, and Ion Stoica. 2025. Moe-lightning: High-throughput moe inference on memory-constrained gpus. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. 715–730.

[13] Shenggan Cheng, Ziming Liu, Jiangsu Du, and Yang You. 2023. ATP: Adaptive Tensor Parallelism for Foundation Models. arXiv preprint arXiv:2301.08658 (2023).

[14] 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 35 (2022), 16344–16359.

[15] Jiangsu Du, Jinhui Wei, Jiazhi Jiang, Shenggan Cheng, Dan Huang, Zhiguang Chen, and Yutong Lu. 2024. Liger: Interleaving Intra-and Inter-Operator Parallelism for Distributed Large Model Inference. In Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 42–54.

[16] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Attention-store: Cost-effective attention reuse across multi-turn conversations in large language model serving. arXiv e-prints (2024), arXiv–2403.

[17] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. Proceedings of Machine Learning and Systems 6 (2024), 325–338.

[18] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. arXiv preprint arXiv:2407.21783 (2024).

[19] Jiaao He and Jidong Zhai. 2024. Fastdecode: High-throughput gpu-efficient llm serving using heterogeneous pipelines. arXiv preprint arXiv:2403.11421 (2024).

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. 103–112.

[21] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. 2024. Pre-gated moe: An algorithm-system co-design for fast and scalable mixture-of-expert inference. In 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). IEEE, 1018–1031.

[22] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. arXiv:2309.14509 [cs.LG] https://arxiv.org/abs/2309.14509

[23] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. Ragcache: Efficient knowledge caching for retrieval-augmented generation. arXiv preprint arXiv:2404.12457 (2024).

[24] 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 the 29th Symposium on Operating Systems Principles. 611–626.

[25] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. 2023. Accelerating distributed {MoE} training and inference with lina. In 2023 USENIX Annual Technical Conference (USENIX ATC 23). 945–959.

[26] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. Terapipe: Token-level pipeline parallelism for training large-scale language models. In International Conference on Machine Learning. PMLR, 6543–6552.

[27] Yunkai Liang, Zhangyu Chen, Pengfei Zuo, Zhi Zhou, Xu Chen, and Zhou Yu. 2025. Injecting Adrenaline into LLM Serving: Boosting Resource Utilization and Throughput via Attention Disaggregation. arXiv preprint arXiv:2503.20552 (2025).

[28] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, et al. 2024. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. arXiv preprint arXiv:2401.02669 (2024).

[29] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring attention with blockwise transformers for near-infinite context. arXiv preprint arXiv:2310.01889 (2023).

[30] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. 2024. Optimizing llm queries in relational workloads. arXiv preprint arXiv:2403.05821 (2024).

[31] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. 2023. Specinfer: Accelerating generative large language model serving with tree-based speculative inference and verification. arXiv preprint arXiv:2305.09781 (2023).

[32] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In Proceedings of the 27th ACM Symposium on Operating Systems Principles. 1–15.

[33] 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 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). IEEE, 118–132.

[34] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2024. vattention: Dynamic memory management for serving llms without pagedattention. arXiv preprint arXiv:2405.04437 (2024).

[35] 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 25). USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[36] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[37] Noam Shazeer. 1911. Fast transformer decoding: One write-head is all you need, 2019. URL https://arxiv. org/abs (1911).

[38] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. Advances in neural information processing systems 31 (2018).

[39] 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.

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, et al. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. CoRR abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[41] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. 2023. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In Proceedings of the 37th International Conference on Supercomputing. 203–214.

[42] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. 2024. Shadowkv: Kv cache in shadows for high-throughput long-context llm inference. arXiv preprint arXiv:2410.21465 (2024).

[43] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023).

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. Advances in neural information processing systems 30 (2017).

[45] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. 640–654.

[46] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. 2024. Qwen2 Technical Report. arXiv:2407.10671 [cs.CL] https://arxiv.org/abs/2407.10671

[47] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In Proceedings of the Twentieth European Conference on Computer Systems. 94–109.

[48] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). 521–538.

[49] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. Advances in Neural Information Processing Systems 36 (2023), 34661–34710.

[50] 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 24). 193–210.

[51] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. 2024. Nanoflow: Towards optimal large language model serving throughput. arXiv preprint arXiv:2408.12757 (2024).

[52] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. 2025. MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism. arXiv:2504.02263 [cs.DC] https://arxiv.org/abs/2504.02263