Cauchy: A Cost-Efficient LLM Serving System through Adaptive Heterogeneous Deployment

Yihui Zhang[1], Han Shen[2], Renyu Yang[1†], Di Tian[2], Yuxi Luo[1], Menghao Zhang[1], Li Li[1], Chunming Hu[1], Tianyu Wo[1], Chengru Song[2], Jin Ouyang[2]
[1]Beihang University [2]Kuaishou Inc.

ABSTRACT

Recent advances in large language models (LLMs) have intensified the need for serving LLMs that are cost-efficient and QoS-guaranteed. Existing frameworks often co-locate computationally distinct prefill and decode instances on homogeneous GPUs, overlooking their unique resource demands and under-utilizing heterogeneous GPUs. This leads to suboptimal resource utilization and increased capital expenditure. We present Cauchy, a LLM serving framework that adaptively deploys prefill and decode computation to the most suitable heterogeneous GPUs and dynamically schedules user requests. At the core of Cauchy is choosing proper *GPU Combo*, a conceptual GPU combination encompassing diverse GPU configurations, for their cost efficiency in running prefill-decode pairs. Cauchy deploys a set of combos to satisfy QoS requirements (e.g., goodput) of LLM inference. Cauchy further employs hierarchical scheduling to handle user requests, using opportunistic scheduling within the allocated *GPU Combos* and a goodput-weighted round-robin policy across *GPU Combos*. Dynamic autoscaling is used to stabilize the cost-efficiency in the face of surging requests. Experiments show that Cauchy achieves up to a 38.3% improvement in Tokens/USD efficiency over the state-of-the-art baselines, while maintaining strict Service Level Objectives (SLOs). Our work highlights the importance of leveraging workload and GPU heterogeneity to achieve superior cost-efficient LLM serving.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Cloud computing**.

KEYWORDS

Heterogeneous Deployment, Prefill-Decode Disaggregation, LLM Serving

ACM Reference Format:
Yihui Zhang, Han Shen, Renyu Yang, Di Tian, Yuxi Luo, Menghao Zhang, Li Li, Chunming Hu, Tianyu Wo, Chengru Song, Jin Ouyang. 2025. Cauchy: A Cost-Efficient LLM Serving System through Adaptive Heterogeneous Deployment. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3772052.3772264

Corresponding Author: Renyu Yang (renyuyang@buaa.edu.cn).

SoCC '25, November 19–21, 2025, Online, USA

ACM ISBN 979-8-4007-2276-9/25/11…$15.00
https://doi.org/10.1145/3772052.3772264

1 INTRODUCTION

Large language models (LLMs) have significantly revolutionized various aspects of engineering and science, from chatbot to document summarization and code generation [4, 22, 25]. Transformer-based models [1, 20, 21] have become the backbone architecture of modern AI services. Deploying LLM models in a cost-efficient manner remains an unsettled challenge, considering the workload diversity and GPU heterogeneity.

LLM workloads exhibit distinct execution characteristics between prefill and decode phases, broadly falling into three categories: Long Input-Short Output (e.g., document summarization [4]), Balanced Input-Output (e.g., chatbot [25]), and Short Input-Long Output (e.g., creative writing [22]). For example, document summarization tasks take long prompts as input yet produce short outputs, and are thus heavily prefill-dominated. In contrast, creative writing tasks are decode-dominated, as they generate lengthy output sequences from short input prompts. The diversity indicates that the model deployment should be adaptive to different computational and memory demands. However, existing deployment policies [28, 32, 33] are agnostic to such workload patterns and fall short in resource optimization among GPU devices. Recent prefill-decode (PD) disaggregation approaches [26, 27, 37] optimize PD resources to match workload patterns, but their static and homogeneous configurations cannot adapt to runtime variations, leading to suboptimal QoS under dynamic traffic conditions.

Meanwhile, modern heterogeneous GPUs exhibit varying cost and performance characteristics. As shown in Table 1, NVIDIA H800 GPU offers superior arithmetic efficiency (TFLOP/USD) ideal for prefilling, while NVIDIA H20 GPU provides higher memory bandwidth efficiency (GB/USD) suitable for decoding. The state-of-the-art prefill-decode disaggregation schemes [26, 27, 37] simply aim to improve goodput without elaborating heterogeneity. Mélange [14] is one of the first attempts to advance heterogeneous GPU allocation for LLM serving. It examines the impact of varying configurations including request size, request rate and SLO on the cost efficiency of different GPU devices, and derives the minimal-cost allocation through solving the cost-aware bin packing problem. Mélange focuses on GPU selection based on the holistic performance profiling of fine-grained request configurations, without analyzing prefill/decode phases in depth or accounting for the individual impact of heterogeneous GPUs on the cost efficiency. GPU heterogeneity unleashes a huge potential for optimizing resource allocation by matching the prefill-decode instances with the most suitable and cost-efficient GPUs.

Table 1: GPU Specifications (Data collected from [11, 34] specifications as of June 2025)

GPU	TFLOPs	BW (GB/s)	Mem (GB)	Price ($/h)	TFLOPs/BW	TFLOP/$	GB/$	Recommended Deployment
H800-SXM	989	3350	80	2.69	0.30	1.32M	4.48M	**Prefill**
A10	125	600	24	0.75	0.21	600K	2.88M	**Prefill**
RTX4090	165	1008	24	0.69	0.21	861K	5.26M	**Aggregated**
A800-PCIe	312	1935	80	1.19	0.16	944K	5.85M	**Aggregated**
MI210	181	1638	64	1.40	0.11	465K	4.21M	**Decode**
H20-NVL	148	4000	96	1.50	0.04	355K	9.60M	**Decode**

In this paper, we present Cauchy, an adaptive GPU scheduling and LLM serving framework that navigates the complexity of deploying LLM services through adaptively assigning GPU combinations to different LLM workloads and dynamically scheduling user requests across these heterogeneous GPUs. The key insight is to investigate the distinct performance of different workloads on various combinations of heterogeneous GPU devices, and select the most suitable combinations that can maximize the token throughput per monetary cost while satisfying the QoS such as goodput. This design is driven by the fact that GPU clusters across different environments – from cost-sensitive clouds to resource-constrained organizations – are typically heterogeneous and have limited number of each GPU type. Cauchy introduces *GPU Combo* as the logical representation of GPU combination and the basic unit of scheduling, and automatically allocates GPUs by selecting top *GPU Combos* and determines the optimal number of each *GPU Combo* via multi-objective optimization. Cauchy employs hierarchical scheduling to handle requests, using opportunistic scheduling within the allocated *GPU Combos* and a goodput-weighted round-robin policy across *GPU Combos* to minimize GPU idle time. Dynamic autoscaling is performed to stabilize the cost-efficiency and service quality in the face of surging requests. Experiments on real-world datasets demonstrate the effectiveness of Cauchy. Compared to heterogeneity-agnostic deployment schemes, Cauchy achieves up to a 38.3% improvement in Tokens/USD efficiency. Cauchy reduces end-to-end latency by up to 59.1%, while maintaining high goodput and consistent performance across different workloads.

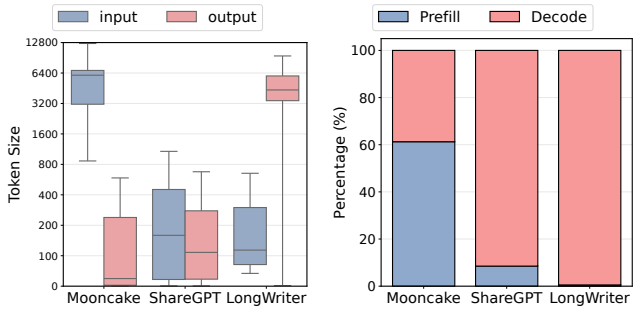
The key contributions of this paper are as follows:

- A cost-efficiency optimization framework that estimates and validates the optimal deployment of heterogeneous GPUs across diverse workloads, maximizing token throughput per monetary cost while satisfying QoS requirements (§3.2 and §3.3).
- A hierarchical scheduling architecture that optimizes request distribution across heterogeneous GPU devices to maximize system goodput and minimize latency (§3.4).
- An elastic autoscaling mechanism that dynamically adjusts *GPU Combo* deployments in response to workload fluctuations, maintaining high cost-efficiency and goodput (§3.5).

2 BACKGROUND

2.1 LLM Serving

LLM Inference. LLM inference consists of two distinct phases: prefill phase and decode phase. In the prefill phase, the model processes the entire input sequence in parallel, computing hidden states and attention scores across all token pairs. This phase is compute-intensive, involving heavy matrix multiplications and softmax operations, thus requiring high arithmetic efficiency (TFLOP/USD). In contrast, the decode phase generates tokens sequentially, one at a time, relying heavily on previously computed KVCache to perform attention over the entire token history. This phase is memory-intensive, demanding high memory bandwidth efficiency (GB/USD). The different computational characteristics of these two phases necessitate distinct optimization strategies and cost considerations.



(a) Length Distribution **(b) Time Distribution**

Figure 1: Heterogeneity between different workloads a) input and output distribution b) prefill and decode time of different workloads under NVIDIA H800-SXM testbeds with Llama-3.1-8B [13].

Prefill-Decode Aggregation or Disaggregation. There are many works [2, 16, 35] that aggregate prefill and decode phases on the same GPU. This approach avoids intermediate state transfer across GPUs, as the KVCache generated during the prefill phase can be directly used by the subsequent decode phase. This scheme is particularly beneficial for long inputs where the KVCache size is substantial. Loading model weights multiple times across different instances can be eliminated, thereby conserving GPU memory.

Disaggregation schemes [26, 27, 37], on the other hand, separate the prefill and decode phases onto different GPUs, thereby enabling hardware selection and optimization strategies tailored to the characteristics of each phase. Intuitively, GPUs with higher compute capabilities can be dedicated to prefill instances, while GPUs with higher memory bandwidth can be allocated to the decode phase. However, disaggregation incurs overhead from KVCache migration, especially with long-context inputs, where the KVCache size is substantial. It is therefore imperative to adaptively tune the prefill-decode configuration and choose the appropriate architecture based on the workload characteristics.

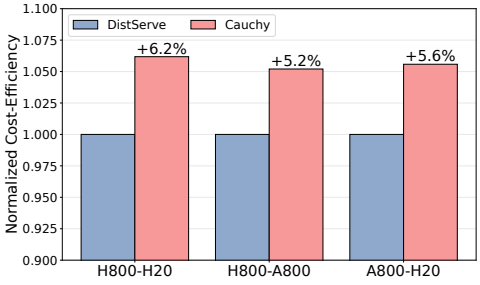


Figure 2: Heterogeneous vs. Homogeneous Disaggregation: in H800-H20 setup, Cauchy dedicates 2 H800 to prefill and 2 H20 to decode, while homogeneous baseline (DistServe) deploys 2 instances: (1 H800 prefill + 1 H800 decode) and (1 H20 prefill + 1 H20 decode). Both deployments use exactly 2 H800 and 2 H20 GPUs in total, with results normalized to DistServe.

2.2 Workload Diversity

As shown in Fig. 1, representative LLM workloads can be roughly categorized according to the length of their input and output.

- **Long Input-Short Output.** Services like summarization [4] process long inputs but produce brief outputs, making them prefill-dominated (>60% latency). The large KVCache strains compute during the prefill phase, while the decode phase underuses memory bandwidth. Aggregated architectures are preferred in this context to avoid prohibitive KV cache migration overhead.
- **Balanced Input-Output.** Conversational service [25] handles symmetrical sequences, balancing compute-memory demands. Moderate KVCache allows flexible architectures: aggregated architecture simplifies cache management, while disaggregated one suits high-throughput streaming. Hybrid approaches may dynamically adjust resources based on real-time workload mixes.
- **Short Input-Long Output.** Text and code generation tasks [22] have minimal inputs but lengthy outputs, making them decode-bound (>95% latency). Negligible prefill KVCache reduces transfer overhead, favoring disaggregated architectures to maximize memory bandwidth for memory-bound generation.

As will be shown in our study, the selection of different GPU types for prefill and decode phases has a noticeable impact on the performance of each type of workload.

2.3 Impact on LLM Inference Performance

Industrial LLM serving systems [3, 12, 23] measure cost efficiency (CE) through Tokens/USD - the number of processed tokens per dollar spent. As shown in Table 1, GPUs exhibit intrinsic heterogeneity – H800 delivers 3.7× higher compute efficiency but 2.1× lower memory efficiency than H20-NVL, creating a 7.5 × disparity in compute-to-memory ratio. We conduct an empirical study to showcase the potential for performance improvement by exploiting GPU heterogeneity. We deploy Llama-3.1-8B using the ShareGPT dataset under the same QPS on H800 and H20. Fig. 2 shows it consistently outperforms the disaggregated deployment scheme with homogeneous configurations, achieving over 5% Tokens/USD improvement. Even when the cross-node KVCache overhead becomes noticeable, a heterogeneous deployment (e.g., H800-A800, A800-H20) retains superior cost efficiency.

2.4 Research Requirements

Designing and implementing an elastic LLM serving system for real-world heterogeneous clusters needs to satisfy the following research requirements.

- **Quantifying the performance impact of heterogeneous GPU combinations on LLM workloads.** As hardware divergence has an explicit impact on phase-specific performance, there is a need to optimize GPU allocations for compute-bound prefill and memory-bound decode phases jointly. It is thus desirable to precisely model the cost-efficiency of GPU combinations with heterogeneous devices, thereby unleashing the potential for LLM serving acceleration.
- **Optimizing the holistic cost-efficiency of GPU allocation while adhering to serving QoS.** It is critical to optimize the cost-efficiency by mapping diverse LLM workloads onto heterogeneous GPUs while satisfying economic and performance constraints. The optimization should take into account hardware capabilities and dynamic workload requirements (e.g., varying input/output patterns and SLOs).
- **Stabilizing the LLM serving for surging requests.** User requests exhibit inherent temporal fluctuations. LLM serving systems need to dynamically address the diverse workload types and fluctuating request volumes.

3 OUR APPROACH

3.1 Overview of Cauchy

GPU Combo. To align with the distinct resource requirements of the prefill and decode phases, we introduce the *GPU Combo*, a pair of GPU devices defined by their types and counts that are best suited to each phase, respectively. For instance, <2×H800, 4×H20> denotes a pair of heterogeneous GPUs where two H800 and four H20 devices are allocated to prefill and decode instances. When a phase requires multiple GPUs of the same type, we prioritize consolidating them into a single instance with higher Tensor Parallelism (TP), rather than deploying multiple smaller instances. Heterogeneous GPU combinations demonstrate directionality - assigning the high-FLOP GPU to prefill and high-bandwidth GPU to decode yields superior cost-efficiency, compared with the reverse configuration.

Cost Efficiency. Aligning with widely-used pricing strategies from major cloud providers, we define *Cost-Efficiency* (*CE*) as the number of processed tokens (including input and output) per US dollar spent. *CE* captures both the compute-bound processing of input tokens in the prefill phase and the memory-bound generation of output tokens in the decode phase. *CE* is a higher-is-better metric that reflects the economic effectiveness of the deployment strategy and the resulting improved resource utilization.

Architecture. Fig. 3 illustrates Cauchy's architecture and the basic workflow among components. The system operates in two phases:

- **Deployment Phase**. Upon receiving a LLM service requirement (model configurations, workload pattern, and expected goodput), Cauchy initiates a modeling process to evaluate all feasible *GPU Combos*. This involves estimating the CE of each *GPU Combo* based on the workload's characteristics (§3.2), and retrieving

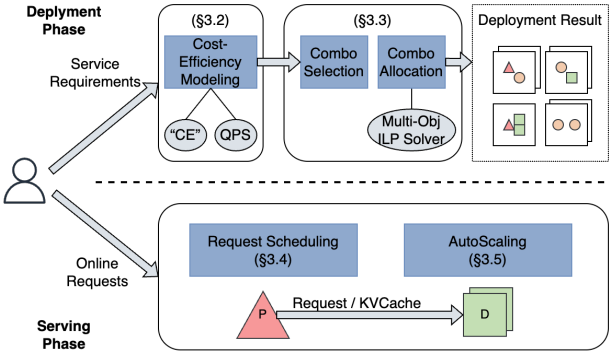


Figure 3: Overview of Cauchy. Shapes represent different GPU types.

pre-profiled goodput for the corresponding workload type. Afterwards, Cauchy generates a set of *GPU Combo* candidates tailored to the workload (§3.3.1). A multi-object integer linear programming (ILP) solver is then employed to determine the optimal number and type of *GPU Combos* to deploy, maximizing CE under cluster resource constraints while satisfying the user's goodput target (§3.3.2).

- **Serving Phase.** Once *GPU Combos* are deployed, incoming requests are routed through a hierarchical scheduler that operates at two levels for load balancing and performance optimization (§3.4): i) inter-combo scheduling, during which requests are distributed across *GPU Combos* using a Goodput-Weighted Round Robin policy for load balancing according to each *GPU Combo*'s capacity; and ii) intra-combo scheduling, during which requests are dynamically forwarded between prefill and decode instances within a *GPU Combo* by using an opportunistic strategy to minimize GPU idle time and reduce the serving latency. Moreover, to handle workload fluctuations, Cauchy conducts combo-level autoscaling, based on real-time QPS monitoring within a sliding time window. When a persistent deviation from the expected load is detected, rescheduling of *GPU Combos* is triggered to maintain goodput while preserving a competitive *CE* (§3.5).

3.2 Cost-Efficiency Modeling

This section describes how to quantify the cost-efficiency of a given *GPU Combo* assigned to a specific LLM workload. The model configurations (e.g., hidden size h, number of layers l) referenced in our modeling, as detailed in Table 2, are used to derive the LLM related coefficients $C_1 - C_4$ in the cost-efficiency model.

3.2.1 Calculating Token Throughput. The theoretical study of token throughput is based on the fact that prefilling is compute-bound and decoding is memory-bound.

Prefill Phase. The prefill phase exhibits quadratic complexity with respect to input length R_{in}, primarily due to attention computations, with an additional linear term from feed-forward networks (FFN). We first define two coefficients:

$$C_1 = 4lh \quad \text{(Quadratic attention term coefficient)} \tag{1}$$

$$C_2 = 8lh^2 + 6lhi \quad \text{(Linear FFN term coefficient)} \tag{2}$$

Table 2: Notation Descriptions

Symbol	Description
l	number of hidden layers
h	hidden size
n	number of attention heads
s	head size ($s = h/n$)
k	number of key-value heads
b	block size of PagedAttention
i	intermediate size
v	vocab size
e	max position embeddings
d	torch dtype
B	batch size
R_{in}	average input tokens per request
R_{out}	average output tokens per request
GPU_n	number of GPUs
GPU_f	peek FLOPs of GPU
GPU_b	peek memory bandwidth of GPU
GPU_p	hosting price of GPU
$S_{prefill}$	input token throughput
S_{decode}	output token throughput

The FLOP requirement is then:

$$FLOP_{prefill} = B(C_1 R_{in}^2 + C_2 R_{in}), \tag{3}$$

The prefill time is bounded by arithmetic performance:

$$T_{prefill}(B, R_{in}) = \frac{B(C_1 R_{in}^2 + C_2 R_{in})}{GPU_n \cdot GPU_f} \tag{4}$$

Hence, the prefill throughput can be calculated by:

$$S_{prefill} = \frac{B \cdot R_{in}}{T_{prefill}(B, R_{in})} = \frac{GPU_n \cdot GPU_f}{C_1 R_{in} + C_2} \tag{5}$$

Decode Phase. To decode R_{out} tokens, we first define two coefficients characterizing the memory access requirements:

$$C_3 = d(2vh + (4h^2 + 3hi + 2h)l) \quad \text{(Static model weight)} \tag{6}$$

$$C_4 = 2dlks \quad \text{(KVCache per token)} \tag{7}$$

The total memory access volume consists of weight access and KVCache transfer:

$$\text{Byte}_{decode} = \sum_{j=0}^{R_{out}-1} [C_3 + B \cdot C_4 (R_{in} + j)] \tag{8}$$

$$= (C_3 + B \cdot C_4 R_{in}) R_{out} + \frac{1}{2} B \cdot C_4 R_{out}^2, \tag{9}$$

The decode time is bounded by memory bandwidth:

$$T_{decode}(B, R_{in}, R_{out}) = \frac{(C_3 + B \cdot C_4 R_{in}) R_{out} + \frac{1}{2} B \cdot C_4 R_{out}^2}{GPU_n \cdot GPU_b} \tag{10}$$

The decode throughput turns out to be:

$$S_{decode} = \frac{B \cdot R_{out}}{T_{decode}(B, R_{in}, R_{out})} = \frac{GPU_n \cdot GPU_b}{\frac{C_4 R_{out}}{2} + C_4 R_{in} + \frac{C_3}{B}} \tag{11}$$

3.2.2 Calculating Cost-Efficiency. We define cost-efficiency (CE) as Tokens/USD, representing the number of processed tokens per dollar spent. The workload pattern for prefill and decode phases is jointly determined by model parameters (C_1-C_4) and request characteristics (R_i, R_o, B). We first define workload-specific coefficients that capture this combined effect:

$$A_1 = \frac{1}{C_1 R_{in} + C_2} \quad \text{(Prefill workload pattern)} \tag{12}$$

$$A_2 = \frac{1}{\frac{C_4 R_{out}}{2} + C_4 R_{in} + \frac{C_3}{B}} \text{ (Decode workload pattern)} \tag{13}$$

The cost-efficiency is then:

$$CE = \frac{S_{prefill}}{GPU_{n_{prefill}} \cdot GPU_{p_{prefill}}} + \frac{S_{decode}}{GPU_{n_{decode}} \cdot GPU_{p_{decode}}} \tag{14}$$

Substituting the throughput expressions from Eq. 5 and Eq. 11, the cost-efficiency becomes:

$$CE = A_1 \cdot \frac{GPU_{f_{prefill}}}{GPU_{p_{prefill}}} + A_2 \cdot \frac{GPU_{b_{decode}}}{GPU_{p_{decode}}} \tag{15}$$

Equation 15 reveals the following key insights:

- **Relative Advantage Principle**. Cauchy's efficiency stems from aligning each phase with the GPU that holds a comparative advantage in the required resource per dollar. Each phase is assigned to the type of GPU that holds a relatively higher ratio of the required resource (TFLOPs for prefill, bandwidth for decode) per dollar, when compared with other available GPUs.
- **Workload-Dependent Coefficients**. The weights A_1 (prefill) and A_2 (decode) are dynamically determined by workload characteristics. This explains why the *GPU Combo* A800-H20 excels for short-input-long-output workloads (high A_2 dominance) but underperforms for long-input-short-output workloads (A_1 dominates but bandwidth remains underutilized).

3.3 Combo-Based GPU Allocation

Cauchy allocates GPUs in units of *GPU Combos* to meet the paired requirements of prefill and decoding instances on a per-workload basis. GPU allocation is performed in two steps: selecting the *GPU Combos* most likely to have the highest cost-efficiency, and determining the optimal number for each selected *GPU Combo*. Runtime workload characteristics and SLO constraints can automatically navigate the optimal *GPU Combo* selection and deployment.

3.3.1 Adaptive Combo Selection. Cauchy shortlists the *GPU Combo* candidates from all possible combinations primarily based on their individual *CE* for each workload scenario. The procedure encompasses the following steps:

- **Workload Estimation**: Computing the phase-wise coefficients (A_1, A_2) from model configurations and input/output token lengths.
- **Profiling**: For each *GPU Combo*, calculating the *CE* when conducting a given LLM inference – considering such influencing factors as FLOPs, bandwidth, and price – and profiling goodput under given SLA.
- **Pareto Filtering**: Retaining the optimal configuration for each unique GPU pair, eliminating suboptimal directional variants (e.g., keeping H800-H20 while discarding H20-H800 when the former has higher *CE*).
- **Candidate Ranking**: Sorting valid configurations in descending order of CE. The candidate information typically includes various *GPU Combos* with their measured cost-efficiency metrics and the actual goodput performance.

We also generalize the *GPU Combo* concept to include combinations of only homogeneous devices, to accommodate some LLM workloads suitable for prefill-decoding aggregation. This combo selection criterion balances cost and performance trade-offs, ranging from cost-efficient high-CE options to high-performance configurations. It maintains deployment flexibility by preserving different *GPU Combos* for various SLO requirements, especially in GPU clusters with fluctuating saturation levels.

3.3.2 Combo Deployment Optimization. We formalize the deployment as an optimization problem to determine the specific *GPU Combo* allocation for each submitted LLM workload, aiming for maximizing holistic cost-efficiency while adhering to resource constraints and ensuring LLM QoS.

Problem Formulation. Cauchy takes the candidate information (§3.3.1) and the user requirements as inputs. User requirements are specified as follows: i) the workload type (as categorized in §2.2), and ii) the target goodput, defined as the SLO-satisfying throughput that inherently accounts for latency requirements.

The optimization aims to minimize deployment costs while maximizing cost-efficiency. Since the most cost-efficient *GPU Combos* are often more expensive to deploy, Cauchy manages this trade-off by selecting combos that meet goodput requirements at the lowest cost, even if they have marginally lower cost-efficiency. For instance, to satisfy a residual goodput of 0.80, it chooses an A800 Combo (goodput=0.82 req/s, price=1.09 $/h, CE=1.07M token/$) over an H20 Combo (goodput=1.23 req/s, price=1.50 $/h, CE=1.15M token/$), as the former fulfills the requirement at lower cost.

The system must satisfy the goodput requirements for each model to maintain system performance and user expectations. Additionally, the total GPU resources consumed across all deployments cannot exceed the available cluster capacity. These constraints are critical to ensure that the optimization process is practical and feasible within the given resource limitations.

Specifically, the multi-objective optimization problem can be formulated with the workload input D_m:

$$\begin{aligned} \min_{\mathbf{x} \in \mathbb{Z}^+} \quad & \sum_{n=1}^{N} \sum_{m=1}^{M} \frac{P_n}{C_n} \cdot x_{mn} \\ \text{s.t.} \quad & \mathcal{T}_m(\mathbf{x}_m) \geq \mathcal{D}m, \qquad \forall m \in 1, \ldots, M \\ & \sum_{m=1}^{M} \mathcal{R}_g(\mathbf{x}_m) \leq G_g, \qquad \forall g \in \mathcal{G} \end{aligned} \tag{16}$$

where the primary decision variable x_{mn} represents the number of *GPU Combo* type n allocated to model m. The objective function combines both cost minimization and efficiency maximization through the composite term P_n/C_n, where P_n denotes the hourly price of *GPU Combo* n and C_n represents its cost-efficiency (Tokens/USD). The constraints ensure that the goodput demand $\mathcal{D}_m$ is satisfied for each model m by the goodput function $\mathcal{T}_m(\mathbf{x}_m)$, and that the total GPU allocation across all deployments remains within the cluster capacity G_g for each GPU type g.

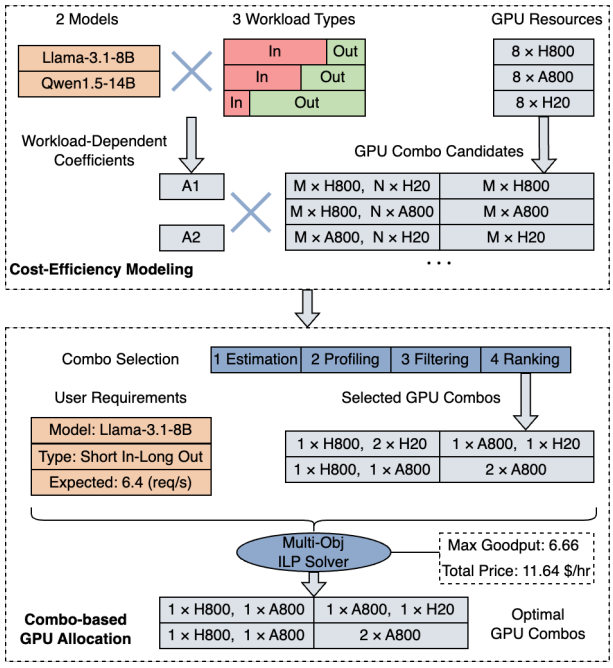


Figure 4: An example of CE modeling and combo-based allocation.

Problem Solver. Cauchy solves the problem by using the PuLP[8] package in Python. The GPU Combo search space grows as $O(G^2)$ for G GPU types, encompassing both homogeneous and heterogeneous combinations with different prefill-decode counts (from 1:1 to 2:6). In our experimental setup with 3 GPU types, this yields up to 12 candidate combos, and the solver typically converges within 20-50 milliseconds. When the optimal solution cannot be obtained within the specified constraints, Cauchy resorts to a fallback strategy – relaxing the optimization objective of maximizing cost-efficiency. This prioritizes the satisfaction of user QoS requirements over the strict optimization of cost-efficiency.

3.3.3 Working Example. Fig. 4 illustrates the evaluation and allocation of different *GPU Combos* using cost-efficiency modeling and goodput profiling. We consider a cluster with 24 heterogeneous GPUs (8 H800s, 8 A800s, 8 H20s), on which we evaluate Llama-3.1-8B [13] and Qwen1.5-14B [6] using three datasets (CNN DailyMail [15, 30], ShareGPT [18], LongWriter-6K [7]) that cover all workload types.

For each model–dataset pair, Cauchy derives the phase-specific coefficients A_1 and A_2 from workload statistics and model configurations, enumerates all feasible *GPU Combos* among H800, A800, and H20, and computes their theoretical *CE* via Eq. 15. Offline profiling under SLOs (TTFT ≤ 10/5/1 s; TBT ≤ 50/30/30 ms) populates an allocation lookup table with CE-goodput pairs. When a user submits a requirement for Llama-3.1-8B on a short input-long output workload with goodput ≥ 6.4 req/s, the Combo Selector filters unsuitable options and ranks the remainder by CE. The top four *GPU Combo* candidates – <1×H800, 1×A800>, <1×A800, 1×H20>, <2×A800>, and <1×H800, 1×H20> – are passed to the ILP solver. Under the cluster's resource constraints, the solver produces the final deployment plan: two <1×H800, 1×A800> combos, one <1×A800, 1×H20> combo, and one <2×A800> combo.

3.4 Hierarchical Request Scheduler

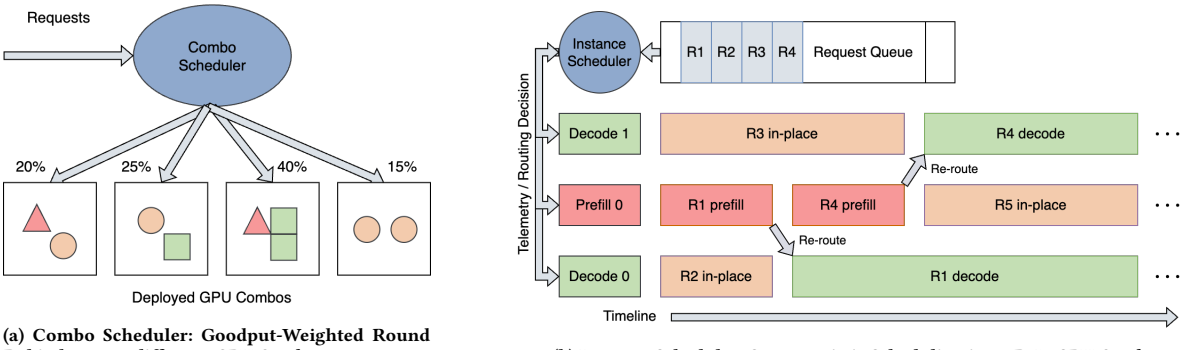
Cauchy employs a hierarchical LLM serving architecture to optimize request distribution across the allocated GPU devices. To ensure efficient resource utilization while maintaining stringent latency SLOs, the serving framework consists of the following components: i) serving components such as API Gateway for service-level routing and request classification, and QPS Monitor for autoscaling triggers; ii) scheduling components including Combo Scheduler for inter-combo load balancing and Instance Scheduler for intra-combo optimization. The key techniques are as follows:

3.4.1 Combo Scheduler for Inter-Combo Balancing. As shown in Fig. 5a, Combo Scheduler employs a *Goodput-Weighted Round Robin* policy to distribute requests across *GPU Combos* in proportion to their profiled goodput capacities (§3.2). Each *GPU Combo*'s weight is dynamically calculated using its goodput divided by the service's total goodput, ensuring performance-proportional allocation while respecting SLO constraints. This can automatically adapt to runtime changes in *GPU Combos* configurations and workload characteristics, thereby maintaining optimal load balancing across heterogeneous hardware. By using goodput as the weighting metric, the scheduler intrinsically prevents over-subscription of any single *GPU Combo*, enabling Cauchy to maximize the utilization of compute and memory capacity across different GPU combinations.

The goodput-weighted distribution provides inherent fairness: high-performance *GPU Combos* handle proportionally more requests without being overwhelmed, while lower-performance ones contribute available goodput without becoming bottlenecks. This balancing mechanism is particularly effective for heterogeneous deployments where different *GPU Combos* can span order-of-magnitude capability differences. The scheduler continuously maintains this balance by adjusting the weights in real time as new *GPU Combos* are added or removed during autoscaling (§3.5).

3.4.2 Instance Scheduler for Intra-Combo Optimization. Ideally, with unlimited GPU resources, Cauchy could always select the optimal *GPU Combo* with perfectly balanced prefill-decode instances ratio to match various workloads' characteristics. However, real-world constraints including dynamic variations in input-output distributions and limited GPU availability oftern force suboptimal deployments, preventing prefill and decode instances from maintaining a balanced KVCache production and consumption. Moreover, traditional round-robin scheduling compounds this problem by rigidly assigning requests to prefill-then-decode paths. This leaves prefill GPUs underutilized while making decode instances the performance bottleneck. Cauchy resolves this through Opportunistic Scheduling, a strategy designed to be fully compatible with the continuous batching[35]. In particular, it employs three collaborative techniques.

- Continuous telemetry collection via lightweight ping probes that monitor per-instance metrics—including remaining tokens and pending request queues—across all instances;



(a) Combo Scheduler: Goodput-Weighted Round Robin between different *GPU Combos*

(b) Instance Scheduler: Opportunistic Scheduling in a 1P2D *GPU Combo*

Figure 5: Hierarchical request scheduling example

- Dynamic workload redistribution that redirects incoming requests to currently underutilized instances (either prefill or decode) when telemetry detects idle capacity. These requests are immediately incorporated into the target instance's continuous batch for processing, avoiding any synchronization overhead;
- Runtime path rewriting that replaces default forwarding rules (e.g., modifying prefill-to-decode mappings from "0→0" to flexible "0→null" markers), enabling full in-place execution without cross-instance transfers.

This metadata-driven approach routes and executes disaggregated workflows locally through microsecond-level path switching. This eliminates GPU idle time without costly context switches or physical data movement, while preserving isolation guarantees. Fig. 5b illustrates how Opportunistic Scheduling strategy works in a 1P2D *GPU Combo*. Initially, Prefill 0 processes the first batch while Decode 0 and 1 remain idle. The scheduler then routes the second and third micro-batches to the idle decode instances for in-place processing. Upon completing prefill, the first batch is assigned to Decode 0, while Prefill 0 begins to process the fourth batch. This scheduling scheme provides request load balancing and can effectively reuse underutilized instances by eliminating GPU idle cycles, thereby maintaining goodput under fluctuating workloads.

3.5 Adaptive Combo-Level Autoscaler

3.5.1 Reactive Combo-Level Scaling. To maintain QoS while avoiding resource over-provisioning, Cauchy initiates fine-grained scaling operations at the granularity of GPU Combos. When real-time request goodput deviates from the provisioned *GPU Combo* capacity, Cauchy initiates scaling operations based on a sliding-window mechanism to handle short-term traffic fluctuations.

Core Idea. The key insight is to implement responsive yet stable scaling through a sliding-window mechanism. This window continuously tracks the current system load, measured in Queries Per Second (QPS), which serves as the trigger for autoscaling decisions. When a throughput mismatch persists beyond the monitoring window (default 2 minutes), autoscaling is triggered only if: i) the relative throughput deviation exceeds the threshold (default 20%), and ii) cluster resources permit safe adjustment. For scaling-out, it selects the most cost-efficient *GPU Combo* covering the deficit by invoking the ILP solver (§ 3.3.2). For scaling-in, it removes *GPU Combos* with closest excess capacity after a graceful period (default 30s). This approach prevents over-reaction to transient fluctuations while ensuring sub-minute response to actual workload changes.

Algorithm. As shown in Algorithm 1, the autoscaler maintains a circular buffer of recent QPS measurements. When the window duration elapses (Line 7), it compares the window-average QPS with current deployment capacity (Line 10) and checks against the ratio threshold (Line 11). Scaling-out occurs when the QPS deficit exceeds the ratio threshold and cluster resources permit additional *GPU Combo* deployment (Line 12). In contrast, scale-in requires the excess throughput to surpass the same ratio threshold (Line 15), ensuring minimal adjustment impact.

Hyperparameter Setting. Window (2min): This duration is carefully calibrated to the infrastructure's instance lifecycle characteristics, accounting for two key time periods: 90 seconds is required for new inference instances to become operational (including resource provisioning, container initialization, model loading and network connection); 30 seconds are needed for graceful termination of released instances (completing in-flight requests and resource cleanup). The 2-minute window ensures scaling operations complete within one monitoring cycle, preventing overlapping scaling commands while maintaining responsiveness. For latency-sensitive services, this can be reduced to several seconds when using pre-warmed instance pools. A threshold of 0.2 is used for triggering scaling operations. Higher values decrease sensitivity to workload fluctuations while lower values increase responsiveness.

3.5.2 Procative Global Combo Refinement. To maintain cost-efficient operation in the long run, the periodic global combo refinement (Lines 22-27) with a *global_window* (30 mins) is employed. The optimization is triggered if QPS remains stable throughout the window after the last scaling. Upon activation, the controller computes a theoretically optimal cluster-wide allocation (Line 24), then updates *GPU Combos* by reconciling the optimal and current states (Line 25). The adjustments follow a safe two-phase process: provisioning new combos before decommissioning redundant resources to avoid service interruption. This window-based approach provides stability against transient workload fluctuations while progressively optimizing global resource efficiency, balancing responsiveness and computational overhead.

Algorithm 1 Dynamic Combo-Level Scaling

Input:
- Initial $\langle W, Q \rangle$ pair (Workload, QPS)
- Total resources R
- Params: ratio = 0.2, local_win = 2min, global_win = 30min

$P \leftarrow$ best_combo(W, Q)
$t_{last} \leftarrow$ now()
$Q_{hist} \leftarrow$ ring_buffer$(local_win)$
while True **do**
  $q \leftarrow$ current_qps()
  Q_{hist}.push(q)
  if now() $- t_{last} \geq local\_win$ **then**
    $Q_{avg} \leftarrow Q_{hist}$.avg()
    $Q_{total} \leftarrow \sum_{p \in P} p.Q$
    $\Delta \leftarrow Q_{avg} - Q_{total}$
    if $|\Delta| >$ ratio $\cdot Q_{total}$ **then**
      if $\Delta > 0$ **and** avail_resources$(R) \geq \Delta$ **then**
        // Scale-out
        $P \leftarrow P \cup$ best_combo(W, Δ)
      else if $\Delta < 0$ **then**
        // Scale-in
        $P \leftarrow$ remove_combos$(P, |\Delta|)$
      end if
      $t_{last} \leftarrow$ now()
      continue
    end if
    if now() $- t_{last} \geq global\_win$ **then**
      // Stable-period optimization
      $P_{opt} \leftarrow$ best_combo$(W, Q_{hist}$.global_avg()$)$
      $P \leftarrow$ optimize(P, P_{opt})
      $t_{last} \leftarrow$ now()
    end if
  end if
end while

3.6 Other Engineering Considerations

KVCache Migration Optimization. To minimize the latency of transmitting KVCache over TCP/IP networks, we introduce a parallelized pipeline for KVCache transmission between prefill and decode instances. The prefill instance proactively pushes each layer's KVCache through dedicated pipelines, overlapping computation with communication to hide migration overhead. The decode instance buffers received key-value pairs in CPU memory before flushing them to GPU. The buffer persists until the request's first forward pass completes, avoiding redundant migrations for repeated prefixes and reducing access overhead.

Instance Role Switching. The system implements role switching for inference instances via RESTful APIs to enable dynamic resource allocation and fault tolerance. Using endpoints like init, activate, show, and stop, running instances can be reconfigured as prefill instances, decode instances, or restored to default standalone aggregated state by disconnecting the prefill-decode connections. This allows seamless integration within *GPU Combos* with automatic connection handling. The design minimizes transition downtime, supports traffic spike load balancing, and simplifies hardware failure recovery through role redistribution.

Fault Tolerance. The system transmits layers independently with CPU buffering to avoid GPU memory contention during peak loads. The CPU buffer mitigates transmission rate fluctuations between instances. The parallel pipeline design provides redundancy where failed pipelines do not affect others, ensuring system reliability. This approach maintains low latency and KVCache consistency across instances.

4 EXPERIMENTS

4.1 Experiment Setup

Hardware and Software. All experiments are performed in a cluster with 8 NVIDIA H800-SXM GPUs, 8 NVIDIA H20-NVL GPUs and 8 NVIDIA A800-PCIe GPUs. Each server is equipped with 2 GPU devices, which allows a tensor parallelism degree of 2, Intel Xeon Platinum 8352Y CPUs @ 2.20GHz, and 1TB RAM. KVCache migration is implemented via Gloo's TCP backend. The inference engine is based on vLLM[19] v0.6.5, and Cauchy serving framework is implemented on Kubernetes v1.16.0. We use the Llama-3.1-8B[13] model for all experiments to ensure a consistent evaluation baseline.

Methodology and Baselines. We conduct a comprehensive evaluation of Cauchy at both macro and micro levels. We first conduct an end-to-end study to evaluate the overall performance against the state-of-the-art deployment strategy named Mélange[14] (§4.2). Afterwards, we examine the performance improvements contributed by individual system components. Specifically, we analyze the *GPU Combo* deployment strategy's operation (§4.3), before quantifying performance gains from intra-combo instance scheduling (§4.5) and inter-combo scheduling (§4.4). We further investigate the performance contributions of autoscaling and KVCache optimization (§4.6 and §4.7). As fine-grained comparisons have different specific goals, we detail the related baselines in each subsection.

Workloads. We select three distinct workload datasets that represent the spectrum of real-world LLM serving scenarios:

- **CNN DailyMail** [15, 30]: This workload features extended input sequences followed by concise outputs, representing document processing tasks. (mean input/output=702/42 tokens)
- **ShareGPT [18]**: This workload features balanced input-output ratios, typical of chatbot interactions. (mean input/output=290/207 tokens)
- **LongWriter-6K [7]**: This workload exhibits short prompts that trigger lengthy generations, common in creative writing applications. (mean input/output=337/1330 tokens)

Evaluation Metrics. To effectively assess the performance of Cauchy, we consider the following metrics:

- **Cost-Efficiency**: The number of tokens processed per dollar spent, calculated by dividing the total tokens by the total cost of all *GPU Combos*.
- **E2E Request Latency**: The total duration from request submission to final token delivery, covering queuing, inference, and network overhead.
- **Time-to-First-Token (TTFT)**: The delay before the first generated token is received.
- **Time-Between-Tokens (TBT)**: The interval between consecutive tokens during streaming.

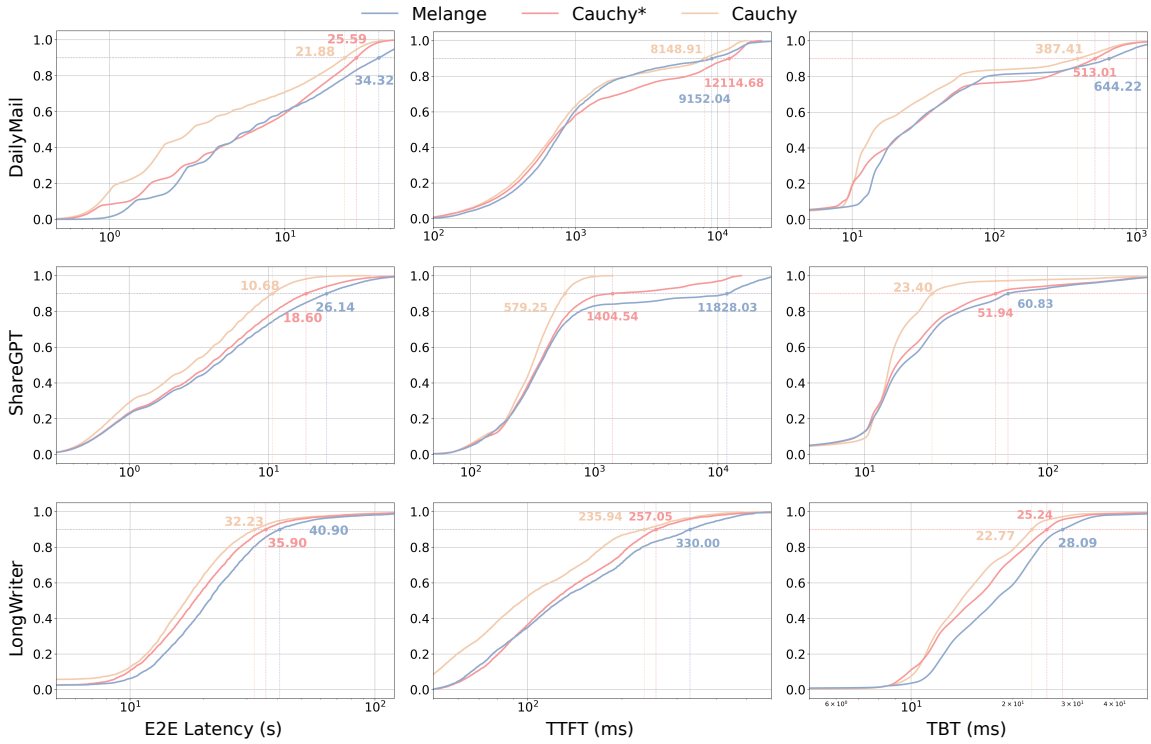


Figure 6: E2E performance comparison between Cauchy and Mélange frameworks across heterogeneous datasets.

Performance Report. To mitigate random fluctuations, we repeat each experiment 20 times and report the average of the collected metrics. We generate requests for each workload using two methods: sampling from the Azure LLM Inference Trace[5, 31] or generating requests following a Poisson distribution, with the same target QPS values (54 for DailyMail, 50 for ShareGPT, and 6 for LongWriter) in both cases. To ensure a fair comparison, the same hyperparameters are used across all competing methods for any given task.

4.2 Overall Performance of Cauchy

Settings. We evaluate Cauchy's performance using Azure LLM Inference Trace[5, 31] across all three workloads.

Comparative Methods. We primarily compare Cauchy with Mélange[14], the most recent work in heterogeneity-aware GPU scheduling, serving as primary baseline. It uses mixed GPUs as aggregated deployments with static allocation. Mélange shares a similar goal of cost optimization, while Cauchy introduces novel innovations in phase-aware *GPU combo* allocation and dynamic scheduling. We omit homogeneous baselines [37] due to the noticeable superiority of heterogeneous deployment over homogeneous approaches shown in §2.3. Baselines are detailed as follows:

- **Mélange**: The state-of-the-art heterogeneity-aware GPU allocation framework, which employs mixed GPUs in aggregated deployments with static allocation.
- **Cauchy ***: A variant of Cauchy that only supports aggregated *GPU Combos*. Compared to Mélange, its primary advantage lies in dynamic resource adjustment.

Results. As shown in Fig. 7, Cauchy * achieves 7.8%-17.1% improvement in cost-efficiency over Mélange, and Cauchy's performance promotes to 16.4%- 38.3% by introducing phase-aware *GPU Combos*. This demonstrates the advantage of matching relative high-compute GPUs to prefill and high-bandwidth GPUs to decode instances, overcoming Mélange's coarse-grained GPU heterogeneity exploitation. As shown in Fig. 6, Cauchy significantly outperforms Mélange, achieving 21.2%-59.1% lower end-to-end latency, 28.5%-95.1% lower TTFT, and 18.9%-61.5% lower TBT. Such consistent improvements reassure Cauchy's ability to maintain QoS across different workload patterns through adaptive scheduling.

4.3 Effectiveness of Allocation Strategy

Settings. To evaluate the effectiveness of Cauchy's *GPU Combo* allocation strategy, we conducted experiments comparing different LP algorithms for deploying *GPU Combo* to handle all three workloads. All allocator methods were evaluated under identical goodput requirements and request arrival patterns.

Comparative Methods. We implemented three linear programming strategies within the Cauchy framework:

- **Maximum Cost-Efficiency LP (MCE-LP)**: This algorithm enforces minimum goodput constraints while maximizing overall cost-efficiency.
- **Minimum Cost LP (MC-LP)**: This algorithm minimizes GPU deployment costs while meeting goodput requirements.
- **Cauchy's Optimized LP (CO-LP)**: This algorithm simultaneously minimizes costs while maximizing cost-efficiency.

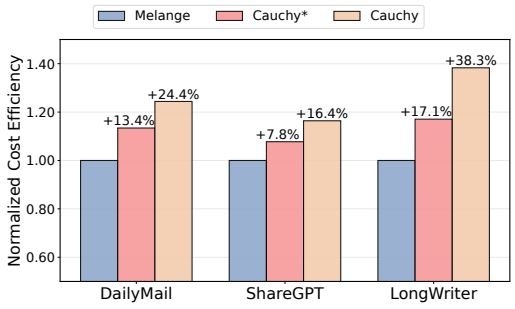


Figure 7: Cost-Efficiency comparison between Cauchy and Mélange.

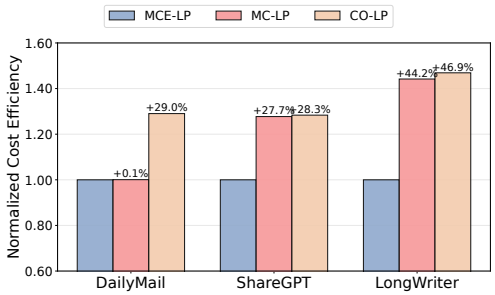


Figure 8: Comparison between different LP formulations.

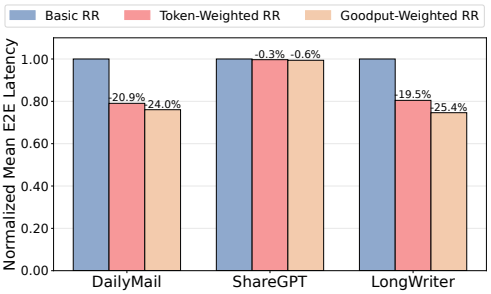


Figure 9: Comparison of *GPU Combo* scheduling strategies with mean E2E request latency metrics. Each scheduling strategy was evaluated under the same QPS across different workloads.

Results. As shown in Fig. 8, CO-LP consistently achieves the highest cost-efficiency across all datasets. On DailyMail, it outperforms MCE-LP by 46.9% by avoiding over-provisioning of high-end GPUs, and maintains a 28.9% cost advantage over MC-LP. While MC-LP's cost-minimization leads to allocations that barely meet goodput thresholds, CO-LP identifies better cost-efficiency trade-offs among similarly-priced options, ensuring optimal *GPU Combos* allocations.

4.4 Effectiveness of Combo Scheduler

Settings. We evaluate inter-*GPU Combo* scheduling using three benchmark datasets deployed on multiple identically configured *GPU Combos*. Workloads follow Poisson arrival processes with request rates matching each dataset's goodput requirements.

Comparative Methods. We implemented three different request scheduling strategies for inter-combo scheduling:

- **Basic Round Robin**: This strategy schedules requests among *GPU Combos* in a cyclic manner.
- **Token-Weighted Round Robin**: This strategy prioritizes scheduling requests to the *GPU Combo* with the highest remaining token count at any given time.
- **Goodput-Weighted Round Robin**: This strategy schedules requests among *GPU Combos* based on the maximum goodput value associated with each *GPU Combo*.

Results. As shown in Fig. 9, Goodput-Weighted RR achieves the lowest latency across all workloads, notably reducing mean latency by 25.4% on LongWriter compared to Basic RR. This gain comes from dynamically balancing load based on each *GPU Combo*'s capacity, unlike Basic RR which ignores its capabilities. While Token-Weighted RR performs similarly on ShareGPT (within 1%), Goodput-Weighted RR consistently outperforms it by 3–6% on DailyMail and LongWriter, showing that token-based scheduling alone cannot fully model combo performance. These results confirm the universal effectiveness of goodput-aware scheduling under diverse workload patterns.

4.5 Effectiveness of Instance Scheduler

Settings. To evaluate the effectiveness of the instance scheduler within disaggregated *GPU Combos*, we conducted experiments on various 1PxD *GPU Combo* configurations (where x = 1, 2, and 3) using the ShareGPT dataset. These configurations used NVIDIA H800 GPUs for the prefill instance and NVIDIA H20 GPUs for the decode instance.

Comparative Methods. We implemented two different request scheduling strategies:

- **Basic Round Robin**: This policy routes requests uniformly across x decode instances in a 1PxD Combo, strictly following the prefill-then-decode paths.
- **Opportunistic Scheduling**: This policy dynamically routes requests to the least-loaded decode instance, while also retaining the option to perform full-phase, in-place inference on any idle instance (prefill or decode).

Results. As shown in Table 3, Opportunistic Scheduling consistently outperforms Basic Round Robin across all configurations. In 1P1D setups, it improves computational efficiency by 8.7% and reduces latency by 8.5% by eliminating prefill GPU idle time through dynamic in-place execution. While the performance gap narrows in multi-decode configurations (1P2D/1P3D) due to improved load balancing, our method maintains a 6-13% advantage across metrics. These results confirm the scheduler's effectiveness in optimizing both balanced and unbalanced workload scenarios.

4.6 Effectiveness of Combo Autoscaler

Settings. To evaluate the benefits of combo-level autoscaling, we initialized all deployments using the same initial allocation plan generated by Cauchy's combo allocator. We used all three datasets and traffic patterns scaled from Azure LLM Inference Trace [5, 31]. We generated dynamic request arrivals while ensuring equal total requests across methods for fair comparison. This setup cleanly isolates the effectiveness of autoscaler's resource adaptation.

Table 3: Performance comparison under different request scheduling strategies

GPU Combo	Strategy	Max CE (K Tokens/USD)	Max Goodput (req/s)	Mean E2E Latency (s)	Mean TTFT (ms)	Mean TBT (ms)
1P1D	Basic RR	1505.23	2.14	22.03	237.03	15.92
	Opportunistic	**1636.24** ↑8.7%	**2.32** ↑8.4%	**20.16** ↓8.5%	**219.09** ↓7.6%	**14.74** ↓7.4%
1P2D	Basic RR	1525.31	2.90	17.75	230.35	12.84
	Opportunistic	**1630.15** ↑6.9%	**3.14** ↑8.3%	**16.38** ↓7.7%	**200.83** ↓12.8%	**12.00** ↓6.5%
1P3D	Basic RR	1231.35	2.96	14.56	221.55	10.51
	Opportunistic	**1329.15** ↑7.9%	**3.22** ↑8.8%	**14.47**	221.10	10.50

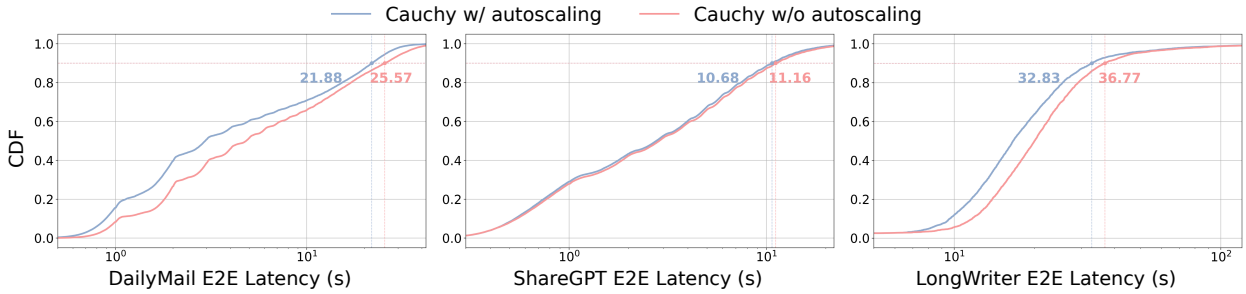


Figure 10: End-to-end (E2E) performance comparison of Cauchy algorithm with autoscale vs. without autoscale across three datasets

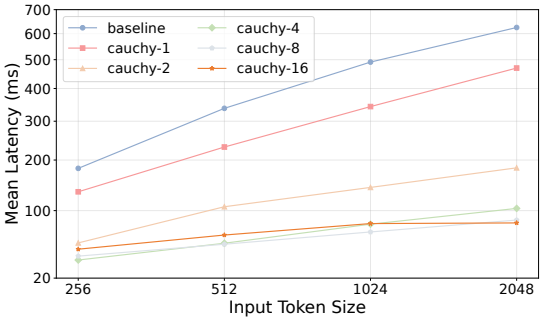


Figure 11: KVCache migration latency between different methods.

Comparative Methods. We compare our combo-level autoscaling against static deployment as the baseline:

- **Cauchy w/o Autoscaling**: This method maintains fixed *GPU Combo* allocation configured for average historical QPS throughout the experiment.
- **Cauchy's Combo-Level Autoscaling**: Cauchy's autoscaler adjusts *GPU Combo* allocations through a dual-phase strategy combining reactive scaling with periodic global optimization.

Results. As shown in Fig. 10, Cauchy's combo-level autoscaling improves end-to-end latency compared to static deployment across all datasets. P90 latency is consistently reduced across all datasets: from 25.57s to 21.88s on DailyMail, 11.16s to 10.68s on ShareGPT, and 36.77s to 32.83s on LongWriter. These improvements are attributed to the autoscaler's dynamic resource adaptation, which avoids both over-provisioning during low utilization and performance degradation during traffic spikes.

4.7 Effectiveness of KVCache Migration

Settings. We evaluate our parallelized KVCache pipelines on NVIDIA H800 and H20 GPUs under cross-rack configurations. Latency $\mathcal{L}(d,t)$ is measured across pipeline degrees d=1–16 and token sizes t=256–2048 at QPS=1. Comparisons include: (1) vLLM's synchronous bulk transfer; (2) Cauchy's serial (degree=1) and parallel (degree=2–16) modes with layer-wise migration.

Results. As shown in Fig. 11, our parallelized KVCache pipeline significantly reduces migration latency compared to synchronous transfer. For 2048-token inputs, latency drops from 625ms to 85ms—a 7.4× improvement—achieved through layer-wise pipelining that overlaps computation and communication. The design maintains 3.4–5.5× speedups across 256–2048 token lengths via dynamic load balancing across pipelines, with optimal performance at pipeline degree=8. Further increasing the degree to 16 shows limited improvement because the cost of managing more threads starts to outweigh the benefits of faster data transfer. This gains stem from three key optimizations: 1) proactive layer pushing through dedicated channels, 2) parallel tranasmission queues for bandwidth maximization, and 3) lightweight synchronization to minimize pipeline coordination overhead.

5 DISCUSSION

Inter-node Networking. Efficient KVCache migration is essential in Cauchy's disaggregated architecture. To hide migration latency by overlapping it with prefill computation, the inter-node bandwidth must meet the constraint:

$$Net \geq \frac{KVCache_{token} \times R_{in} \times GPU_{f_{prefill}}}{FLOP_{prefill}} \tag{17}$$

Substituting Eq. 3 and Eq. 7 yields:

$$Net \geq \frac{C_4 \times GPU_{f_{prefill}}}{B(C_1 R_{in} + C_2)} \tag{18}$$

This requirement is modest: prefilling 2048 tokens for Llama-3.1-8B on an A100 merely requires 10GB/s, and the number is reduced to 5GB/s for the 70B model. Lower Model FLOPS Utilization (MFU), often caused by small batch sizes and inter-operator synchronization, further reduces the bandwidth requirement. This relaxed bandwidth demand makes Cauchy practical in typical data center networks.

Compatibility and Scalability. Cauchy exhibits strong compatibility across diverse hardware and model architectures. Its cost-efficiency optimization is hardware-agnostic, readily incorporating new GPU series from various vendors (e.g., the prefilling-optimized NVIDIA Rubin CPX [24]) to enhance service economics. Cauchy can seamlessly integrate with standard model parallelism techniques to accommodate large-scale models (e.g., 70B or 405B parameters). Similarly, Cauchy can effectively cooperate with expert parallelism and other distributed strategies to support Mixture-of-Experts (MoE) model [20] serving. This indicates the flexibility and generic capability of serving framework for the evolving landscape of LLMs and accelerators.

Performance in Homogeneous Clusters. While Cauchy achieves optimal cost-efficiency in heterogeneous environments, it remains functional in homogeneous clusters with more limited gains. The absence of hardware diversity restricts the system's ability to strategically match GPU capabilities to phase-specific requirements, though scheduling and resource management optimizations still apply.

Limitation. Cauchy's current implementation employs TCP-based transmission for KVCache migration. Several systems like Splitwise [26] have demonstrated the effectiveness of RDMA for reducing latency in disaggregated architectures. Integrating RDMA support represents a key direction for future work. We plan to extend Cauchy to: (1) prioritize forming GPU Combos between RDMA-equipped servers, and (2) implement intelligent request scheduling that routes long-context requests to RDMA-enabled combos to minimize KVCache migration overhead.

6 RELATED WORK

Semi-Disaggregated Architecture. Recent works [9, 17, 29] have explored semi-disaggregated architectures to balance resource utilization and latency. DynaServe [29] introduces a Tandem Serving model that splits requests into virtual sub-requests processed by GPU pairs, enabling elastic load balancing and hybrid execution without rigid disaggregation. semi-PD [17] uses phase-wise disaggregated computation with unified storage and SM-level partitioning to reduce interference and KVCache migration overhead. EcoServe [9] proposes a partially disaggregated strategy that temporally separates phases within instances while coordinating macro-instances for cost-effective scaling. While these systems demonstrate the effectiveness of semi-disaggregation on homogeneous GPUs, Cauchy is orthogonal to them and maintains compatibility, allowing semi-disaggregated instances to be integrated as supplementary *GPU Combos*.

Resource Management. Recent research [14, 28, 32] have explored various strategies to improve LLM serving efficiency. VTC (Virtual Token Counter) [32] ensures fair resource allocation across clients through token-level service tracking. BCA (Batching Configuration Advisor) [28] identifies DRAM bandwidth saturation as the fundamental bottleneck in large-batch LLM inference, and proposes memory-aware batch sizing. Closely related to our work is Mélange [14], which introduces a cost-aware GPU allocation framework that formulates resource assignment as a bin-packing problem. While Mélange effectively leverages GPU heterogeneity, it treats LLM inference monolithically without distinguishing the distinct resource demands of prefill and decode phases, nor does it provide autoscaling mechanisms for dynamic workloads. Cauchy addresses these limitations through phase-aware *GPU Combos* and combo-level autoscaling, enabling more efficient and adaptive resource utilization.

Request Scheduling for LLM Service. Traditional scheduling strategies like First-Come-First-Served suffer from Head-of-Line blocking, limiting throughput and service quality. Advanced scheduling techniques [10, 33, 36] address this through various approaches. Learning-to-rank [10] approach predicts the relative order of LLM request lengths, enabling near-optimal Shortest-Job-First scheduling. Llumnix [33] exploys dynamic request rescheduling and live migration to handle workload heterogeneity. Tempo [36] incorporates SLO awareness through a hybrid approach of conservative estimation and online refinement. These works demonstrate ongoing efforts to enhance LLM service efficiency, fairness, and responsiveness through innovative request scheduling.

7 CONCLUSION

This paper introduces Cauchy, a cost-efficient LLM serving system that fundamentally rethinks GPU resource allocation through adaptive heterogeneous deployment. Our system combines three key innovations: (1) the *GPU Combo*, a novel abstraction that strategically matches hardware capabilities to phase-specific computational demands, (2) a hierarchical scheduling architecture that maximizes resource utilization while maintaining QoS, and (3) an autoscaling mechanism that dynamically adjusts to workload fluctuations. These components work collaboratively to deliver superior cost-efficiency without sacrificing service reliability. We will open-source Cauchy and investigate fine-grained GPU sharing, promoting further advances in cost-efficient LLM serving.

ACKNOWLEDGMENT

We would very much like to thank our shepherd Francisco Romero, and the anonymous reviewers for their valuable comments. Special thanks must go to the overall AI platform team at Kuaishou Inc. and RAIDS Lab team at Beihang University, for their collaborative contribution and countless technical discussion. This work is supported in part by National Key R&D Program of China (Grant No. 2024YFB4505901), in part by the National Natural Science Foundation of China (Grant No. 62402024), in part by the Fundamental Research Funds for the Central Universities, and, last but not the least, by Kuaishou Research Fund. For any correspondence, please refer to the project lead and coordinator Prof. Renyu Yang (renyuyang@buaa.edu.cn).

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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[3] Amazon Web Services. 2025. *Generative AI on AWS.* https://aws.amazon.com/ai/generative-ai Retrieved June, 2025.

[4] Anthropic. 2025. *Ai Document Summary.* https://www.claude.ai Retrieved June, 2025.

[5] Azure Public Dataset. 2025. *AzureLLMInferenceTrace.* https://github.com/Azure/AzurePublicDataset Retrieved June, 2025.

[6] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. *arXiv preprint arXiv:2309.16609* (2023).

[7] Yushi Bai, Jiajie Zhang, Xin Lv, Linzhi Zheng, Siqi Zhu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. LongWriter: Unleashing 10,000+ Word Generation from Long Context LLMs. *arXiv preprint arXiv:2408.07055* (2024).

[8] COIN-OR Foundation. 2025. *A python Linear Programming API.* https://github.com/coin-or/pulp Retrieved June, 2025.

[9] Jiangsu Du, Hongbin Zhang, Taosheng Wei, Zhenyi Zheng, Kaiyi Wu, Zhiguang Chen, and Yutong Lu. 2025. EcoServe: Enabling Cost-effective LLM Serving with Proactive Intra-and Inter-Instance Orchestration. *arXiv preprint arXiv:2504.18154* (2025).

[10] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2024. Efficient LLM Scheduling by Learning to Rank. *arXiv preprint arXiv:2408.15792* (2024).

[11] GetDeploying. 2025. *Cloud GPU Price Comparison.* https://getdeploying.com/reference/cloud-gpu Retrieved June, 2025.

[12] Google Cloud. 2025. *Cost of building and deploying AI models in Vertex AI.* https://cloud.google.com/vertex-ai/generative-ai/pricing Retrieved June, 2025.

[13] 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).

[14] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. 2024. M\'elange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527* (2024).

[15] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching Machines to Read and Comprehend. In *NIPS*. 1693–1701. http://papers.nips.cc/paper/5945-teaching-machines-to-read-and-comprehend

[16] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. 2024. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671* (2024).

[17] 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).

[18] Huggingface. 2025. *ShareGPT datasets.* https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered Retrieved June, 2025.

[19] 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.

[20] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[21] AI Meta. 2024. Introducing llama 3.1: Our most capable models to date, 2024. *URL https://ai. meta. com/blog/meta-llama-3-1/. New models including flagship 405B parameter model, along with upgraded 8B and 70B models featuring 128K context length and multilingual capabilities* (2024).

[22] Microsoft. 2025. *Microsoft Copilot.* https://copilot.microsoft.com Retrieved June, 2025.

[23] Microsoft Azure. 2025. *Azure OpenAI Service pricing.* https://azure.microsoft.com/en-us/pricing/details/cognitive-services/openai-service/ Retrieved June, 2025.

[24] NVIDIA. 2025. *NVIDIA Unveils Rubin CPX: A New Class of GPU Designed for Massive-Context Inference.* https://nvidianews.nvidia.com/news/nvidia-unveils-rubin-cpx-a-new-class-of-gpu-designed-for-massive-context-inference Retrieved Sept, 2025.

[25] OpenAI. 2025. *ChatGPT: smart and simple AI.* https://www.chatgpt.com Retrieved June, 2025.

[26] 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.

[27] 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)*. 155–170.

[28] Pol G Recasens, Ferran Agullo, Yue Zhu, Chen Wang, Eun Kyung Lee, Olivier Tardieu, Jordi Torres, and Josep Ll Berral. 2025. Mind the memory gap: Unveiling gpu bottlenecks in large-batch llm inference. *arXiv preprint arXiv:2503.08311* (2025).

[29] Chaoyi Ruan, Yinhe Chen, Dongqi Tian, Yandong Shi, Yongji Wu, Jialin Li, and Cheng Li. 2025. DynaServe: Unified and Elastic Execution for Dynamic Disaggregated LLM Serving. *arXiv preprint arXiv:2504.09285* (2025).

[30] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1073–1083. https://doi.org/10.18653/v1/P17-1099

[31] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 205–218.

[32] 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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 965–988.

[33] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 173–191.

[34] vast.ai. 2025. *Pricing | Vast.ai.* https://vast.ai/pricing Retrieved June, 2025.

[35] 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.

[36] Wei Zhang, Zhiyu Wu, Yi Mu, Banruo Liu, Myungjin Lee, and Fan Lai. 2025. Tempo: Application-aware LLM Serving with Mixed SLO Requirements. *arXiv preprint arXiv:2504.20068* (2025).

[37] 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.