From Models to Operators: Rethinking Autoscaling Granularity for Large Generative Models

Xingqi Cui
Rice University

Chieh-Jan Mike Liang
Microsoft Research

Jiarong Xing
Rice University

Haoran Qiu
Microsoft Azure Research

Abstract

Serving large generative models such as LLMs and multimodal transformers requires balancing user-facing SLOs (e.g., time-to-first-token, time-between-tokens) with provider goals of efficiency and cost reduction. Existing solutions rely on static provisioning or model-level autoscaling, both of which treat the model as a monolith. This coarse-grained resource management leads to degraded performance or significant resource underutilization due to poor adaptability to dynamic inference traffic that is common online.

The root cause of this inefficiency lies in the internal structure of generative models: they are executed as graphs of interconnected *operators*. Through detailed characterization and systematic analysis, we find that operators are *heterogeneous* in their compute and memory footprints and exhibit diverse sensitivity to workload and resource factors such as batch size, sequence length, and traffic rate. This heterogeneity suggests that the *operator*, rather than the entire model, is the right granularity for scaling decisions.

We propose an ***operator-level autoscaling*** framework, which allocates resources at finer (operator)-granularity, optimizing the scaling, batching, and placement based on individual operator profiles. Evaluated on production-scale traces, our approach preserves SLOs with up to 40% fewer GPUs and 35% less energy, or under fixed resources achieves 1.6× higher throughput with 5% less energy. These results show that the operator, rather than the model, is fundamentally a more effective unit for scaling large generative workloads.

1 Introduction

Serving online inference workloads of large generative models such as large language models (LLMs) and multimodal LLMs is both expensive and performance-sensitive. End users expect fast responses from LLM services, often specified by service-level objectives (SLOs) on time-to-first-token (TTFT) and time-between-tokens (TBT). Cloud providers, in contrast, focus on minimizing GPU cost, improving utilization, and reducing power or energy consumption. Taming this tension is a core challenge in production model provisioning.

Motivation. Intuitively, LLM services can avoid SLO violations by statically provisioning for peak traffic profiles, *i.e.*, sizing the number of model instance replicas to handle extreme cases. However, from experiences at global cloud providers, LLM inference workloads exhibit a high degree of dynamics over time and hard-to-predict input/output sequence lengths; thus, reserved resources can only be utilized at 50% and 39% when provisioning for the P95 demand of text and multimodal workloads, respectively [43, 48, 54]. This implies that static provisioning can incur a significant resource cost.

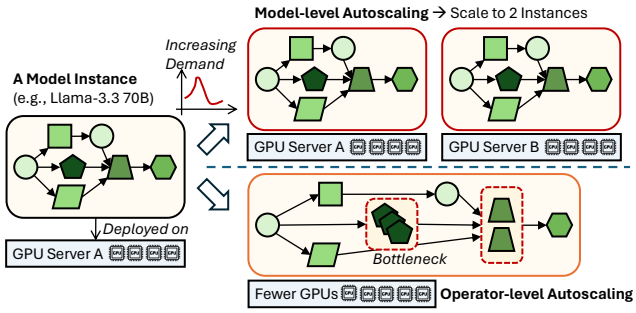


Figure 1: Operator-level vs. model-level autoscaling.

To handle this variability while meeting SLO requirements and achieving efficient GPU usage, the key enabler is SLO-targeted autoscaling of inference clusters. At first glance, it appears that autoscaling can simply be implemented at a ***model-level*** granularity, where the number of model replicas is adjusted to meet SLOs. Despite its simplicity, model-level autoscaling fundamentally limits the ability to meet the two requirements above, as its coarse-grained strategy restricts fine-grained control and adaptability. First, model-level granularity treats the model as one monolithic scaling unit. Generative models today are directed acyclic graphs (DAGs) of heterogeneous *operators* such as attention, linear transformation, and normalization. Effectively, scaling entire model replicas forces all operators to scale uniformly, even when only a small subset contributes to the bottleneck. This *over-provisioning* means that non-critical operators are also replicated to occupy precious GPU cycles, memory, and compute power that could otherwise be shared across workloads.

Second, model-level autoscaling is slow: loading a full model onto additional GPUs incurs significant startup latency (*e.g.*, loading a 70B model takes at least ten seconds on average even with state-of-the-art methods [10]). This delay makes it difficult to adapt quickly to traffic fluctuations that are common in LLM workloads [43], often resulting in SLO violations or costly over-allocation and thus poor utilization.

Our Work. This paper explores a new opportunity for gen-

erative model provisioning: ***operator-level autoscaling*** (as illustrated in Figure 1). Rather than treating the model as one scaling unit, we propose to delve into operator-level granularity, for finer control of GPU resource allocation. The goal is to independently scale the number of replicas for each operator, while trying to meet the model-level SLOs. In addition, scaling at a finer granularity enables fast elasticity, reducing scaling latency from tens of seconds for model-level scaling (*e.g.*, 10s for a 70B model [10]) to sub-second response to handle rapid inference demand changes.

However, operator-level autoscaling raises two novel challenges. First, identifying which operators to scale is non-trivial, and naïvely targeting the slowest operator does not work well: the bottleneck shifts dynamically with workload changes (*e.g.*, QPS, input lengths), and not all operators benefit equally from the same resource scaling. For instance, attention operators are heavily compute-bound and sensitive to sequence length, especially under long context load, while linear operators dominate compute at short sequence lengths but scale less as the sequence length grows. Operators like normalization are lightweight in compute, and benefit disproportionately from batching. Second, scaling operators with isolated device placement wastes resources, whereas colocating scaled operators improves utilization but risks interference across shared GPU resources (SMs, memory, interconnects), necessitating accurate contention modeling.

To address these challenges, we provide theoretical analysis with an SLO-oriented serving framework with operator-level provisioning that (1) identifies scaling candidates that most effectively reduce end-to-end latency through offline profiling on both workload-aware performance sensitivity and resource elasticity analysis, and (2) models colocation contention to guide interference-aware placement. Architecturally, operator autoscaling operates across multiple planes of control: First, the data plane captures the performance characteristics of each operator under diverse workload conditions (batch size, sequence length, query rate); Then, the scaling plane models these profiles, and dynamically computes the scaling plan and configurations for all operators; Finally, the scheduling plane takes the scaling plan, and jointly considers all operator replicas to compute the device assignment plan. This enables more efficient utilization of GPU devices, better SLO preservation, and reduced energy consumption by rebalancing resources across operators to align supply with demand at a finer granularity than model-level autoscaling permits.

Across diverse model architectures (Table 1) on production-scale representative traces [42, 48], operator-level autoscaling achieves the same SLO preservation with up to 40% fewer GPUs and 35% lower energy consumption compared to model-level autoscaling. With prefill-decode disaggregation, the resource savings achieved by operator-level autoscaling are most pronounced in the prefill phase (*i.e.*, 2–3× higher than those of the decode phase), which highlights distinct computational profiles between two stages and suggests that inference clusters can gain substantial efficiency by enabling prefill–decode disaggregation. These results demonstrate that each operator is a fundamentally more effective unit of scaling than the model as a monolith, enabling systems to balance end-user performance requirements with provider efficiency goals more precisely and promptly.

Table 1: Models in characterization study.

Model	Model Size	Modality	Architecture
Qwen2-7B [20]	7B	Text	Dense LLM
Qwen2-MoE [19]	57B (14B)	Text	MoE LLM
Llama3-8B [17]	8B	Text	Dense LLM
Mixtral-8×7B [18]	47B (13B)	Text	MoE LLM
Qwen2.5-VL-32B [21]	32B	Visual	Encoder+LLM

Contributions. We make the first step toward fine-grained resource management at the operator level for efficient generative model serving in cloud datacenters. In summary, our main contributions include:

- Characterization of diverse operator compute–memory sensitivities under varying workload conditions.
- An operator-level autoscaling strategy that exploits operator heterogeneity for dynamic resource allocation.
- A contention-aware operator placement strategy that balances execution efficiency and cost.
- Extensive evaluation with production-scale traces, highlighting improvements in resource efficiency, throughput, and energy consumption across model architectures.

2 Background

2.1 GPU Execution Model for Inference

Computation on GPU is expressed as *kernels*, which run across many threads organized into *blocks* and scheduled to run on *Streaming Multiprocessors (SMs)* in parallel. Each SM contains multiple SM cores—the fundamental compute units responsible for executing kernels. This parallelism is essential for matrix and tensor operations that make up neural network layers in model inference.

For generative models like LLMs, inference consists of a sequence of *operators* such as matrix multiplications, normalization, and attention. These operators form a *directed acyclic graph (DAG)* where edges represent tensor dependencies and nodes correspond to computation kernels. Examples such as residual connections, layer normalizations, and attention branches create multiple dataflow paths within this DAG. However, as outputs depend only on previously computed tensors, the graph remains acyclic. These operators are compiled into GPU kernels and run as part of a GPU *stream*, which is an ordered sequence of kernels that can overlap with data transfers for efficiency [5]. Model weights and intermediate activations reside in GPU memory, while temporary buffers

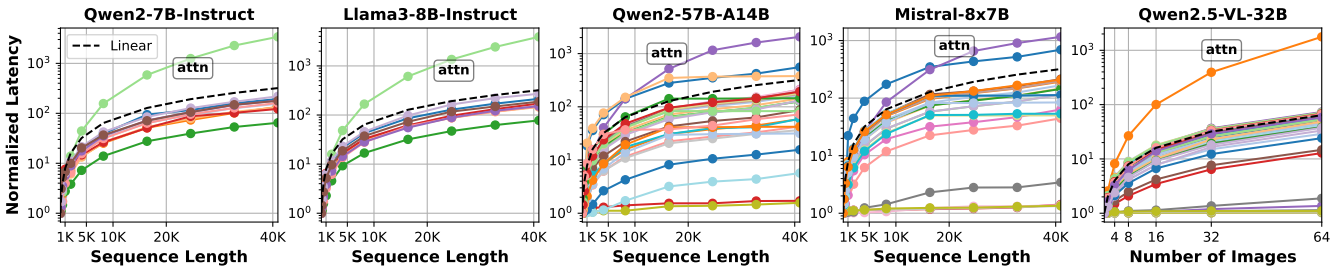


Figure 2: Compute sensitivity to input data sizes, for various operators in different model architectures.

handle transient memory needs during operations. Multiple streams can run concurrently on GPUs to maximize hardware utilization with time or space sharing techniques like MPS [31], MIG [30], and CUDA Green Context [33].

2.2 Generative Model Inference Autoscaling

A generative model inference cluster manages a pool of GPU servers to handle requests for models like LLMs, multimodal models, or diffusion models (*e.g.*, for video generation). These services back diverse applications such as interactive chat [34], deep research jobs with relaxed deadlines [35], or real-time audio streaming [36]. They have varying SLOs and dynamic traffic patterns. Two most common SLOs are on time-to-first-token (TTFT) and time-between-tokens (TBT). This variability creates a fundamental tension: optimal autoscaling must allocate just enough server capacity to preserve user-centric SLOs (especially tail latencies) while minimizing provider-centric goals in cost and energy.

Today's systems rely on ***model-level autoscaling*** that combines *horizontal autoscaling* (*i.e.*, scaling in/out model instances) and *vertical autoscaling* (*i.e.*, scaling up/down model parallelism degrees). Vertical autoscaling tunes intra-replica capacity via tensor or pipeline parallelism based on profiled memory footprint and latency-throughput tradeoffs [48]. Horizontal autoscaling typically combines demand forecasting with queueing-delay controllers and per-model replica scaling, often using signals like tokens-per-sec, queueing delays, and SLO violations [16, 39, 50]. However, short traffic bursts and rapidly varying context lengths frequently cause capacity misalignments and SLO regressions, as provisioning new model replicas is slow. In addition, state-of-the-art autoscalers such as AIBrix [50], DynamoLLM [48], vLLM Production Stack [11], and Chiron [39] fail to take advantage of operator-level heterogeneity for cost saving, i.e., not every operator is equally sensitive to workload changes.

In this paper, we systematically explore the benefits, opportunities, and challenges of ***operator-level autoscaling*** and finer-grained resource management in modern generative model serving clusters.

3 Operator Characterization and Insights

We present the first systematic characterization of the performance and resource characteristics of each individual operator across large generative models. Specifically, we study two dominant model architectures: dense LLMs and mixture-of-experts (MoE)-based architectures. We profile their compute, memory, input/output data volume, and queueing characteristics at the operator level, as well as the impact of GPU resource partitioning. Characterizing these operators provides insights into their *sensitivity* to different system factors such as sequence length, batch size, request arrival rate, and hardware allocation. For each operator, we define sensitivity as the normalized latency relative to a baseline configuration (*e.g.*, batch size of 1 and the shortest sequence length).

Experiment Setup. We conduct experiments on vLLM across diverse models, including Qwen2-7B [20], Qwen2-MoE [19], Llama3-8B [17], Mixtral-8x7B [18], and Qwen2.5-VL [21] (as listed in Table 1), on an Azure GPU server [29] with 8 NVIDIA A100 GPUs. For each model, we evaluate the inference runtime across multiple model configurations, varying prompt lengths, batch size, and tensor parallelism. To characterize performance, we employ the CUDA time profiler to collect GPU kernel runtimes for each operator, based on vLLM's built-in layerwise-profile context manager to instrument the inference execution. This setup captures fine-grained performance metrics for each kernel and operation, enabling a comprehensive analysis of runtime bottlenecks and operator-level summaries, including compute time, weight memory, activation memory, and inter-operator communication volume and data shape. In addition, we leverage NVIDIA MPS [31] to control the allocation of SM cores to individual operators (in the SM sensitivity experiment), and use NVIDIA DCGM [32] to monitor SM utilization at runtime.

Compute Characteristics. We measure compute time as the actual GPU execution time per operator, recorded in microseconds (μs) using CUDA event profiling. This metric isolates the pure computational cost of GPU kernels by excluding CPU overheads and memory-transfer latencies. To understand how operator latency scales under different workloads, we evaluate its *sensitivity* to key generative-model serving parameters

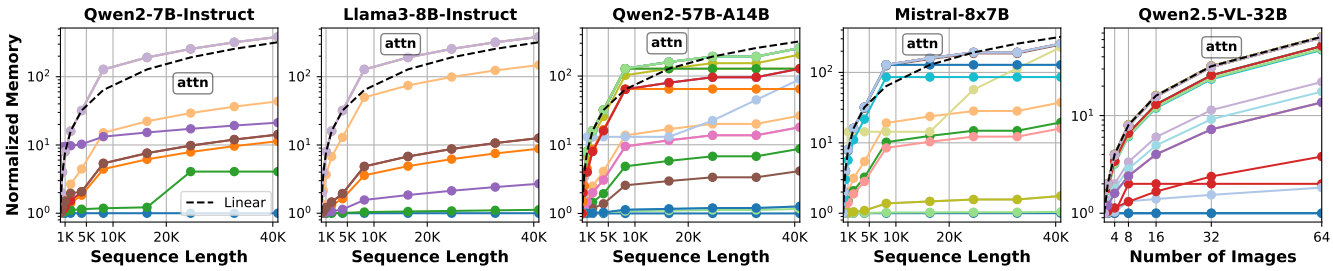


Figure 3: Memory sensitivity to input data size, for various operators in different model architectures.

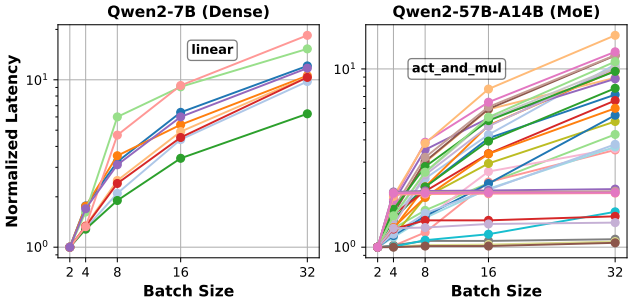


Figure 4: Compute sensitivity to batch sizes.

(*i.e.*, sequence length and batch size). By tracking latency growth with increasing batch size or sequence length, we quantify each operator's scaling behavior for more accurate performance modeling.

Figure 2 shows that the compute sensitivity to sequence length is dominated by the attention operator. In the prefill stage, self-attention exhibits quadratic time complexity with respect to sequence length L (*i.e.*, $O(L^2d)$ with batch size d), since every token attends to all previous tokens. By contrast, other operators—such as feed-forward layers, layer norms, and embedding lookups—scale linearly (*i.e.*, $O(Ld)$) with sequence length and thus show far less increase in normalized latency. Operators like softmax, fill, and sigmoid show nearly flat curves in compute sensitivity to sequence length in MoE models. During decoding, we observe a similar trend across operators, but with smaller slopes, as the cached KV pairs reduce the computational cost of attention. These findings highlight that prefill attention remains the key scaling challenge for generative model serving under long sequence lengths, regardless of model architecture.

Figure 4 shows the compute sensitivity to batch sizes. While most operators exhibit roughly linear scaling with batch size, there is still notable variation in slope across operators, which reflects differences in compute intensity. Operators with heavier per-token arithmetic (*e.g.*, large linear projections, fused MoE layers) scale closer to perfectly linear because their compute dominates memory overhead. In contrast, lighter operators (*e.g.*, layer norms, elementwise activations, small projections) show sub-linear scaling, as fixed kernel launch costs and memory-bound behavior become relatively more significant at larger batch sizes. Thus, even though attention itself becomes linear with batch size, the variation between operators exposes their differing compute-to-memory ratios and can help identify which kernels are most sensitive to batching and which are bottlenecked elsewhere. This reinforces the importance of per-operator profiling rather than assuming uniform scaling across the entire model.

Insight 1: *Operator compute sensitivity varies widely, with attention dominating across model architectures due to quadratic complexity. MoE and Encoder-LLM models exhibit more operators with flat scaling curves.*

Memory Characteristics. Memory footprint is a key constraint for large-model serving, as GPUs must hold not only model parameters but also activations and the key–value (KV) cache during generation. Therefore, memory profiling considers both weight memory and activation memory. Weight memory corresponds to the static storage of model parameters, while activation memory refers to the intermediate tensors and KV cache generated during forward pass computation that depend on request sequence lengths. By monitoring operator memory usage under varying sequence lengths, we quantify each operator's memory sensitivity (similar to compute sensitivity) in terms of how its memory usage scales with workload dimensions to identify memory characteristics.

In transformer models, attention operators dominate memory growth due to their $O(L^2)$ scaling with sequence length, while most other operators grow roughly linearly. However, with FlashAttention [7], the attention operator has linear memory complexity to sequence length due to I/O-aware optimizations. Therefore, we observed that `act_and_mul` fused kernel, together with other linear kernels, have a similar growth trend compared to the attention operator across all models in Figure 3. Lightweight operators like index-select and activation kernels show flatter growth.

Combining compute and memory sensitivity, Figure 5 shows that, for a layer of the Llama2-7B model, some operators are primarily memory-intensive (*e.g.*, norm), while others are primarily compute-intensive (*e.g.*, reshape and cache). Certain operators, such as attention, are intensive in both di-

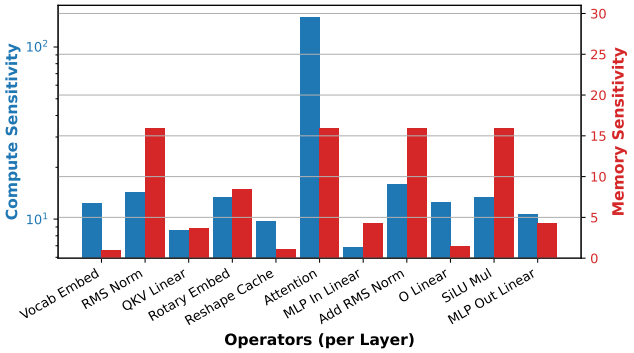


Figure 5: Compute- vs. memory-sensitive operators.

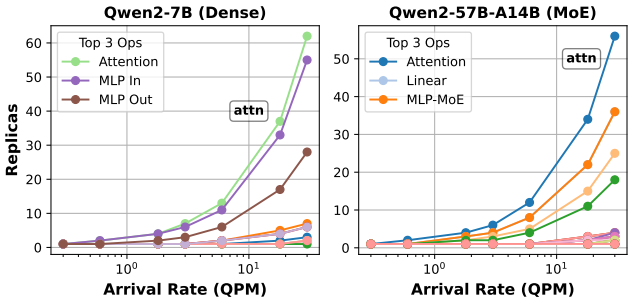


Figure 6: Queueing sensitivity to arrival rate.

mensions. This suggests that different scaling strategies are required depending on the operator's resource profile.

> **Insight 2:** *Memory sensitivity is more evenly distributed across operators compared to compute sensitivity, consistent across model architectures. The two dimensions are uncorrelated—operators may be intensive in both, one, or neither. Unlike compute, memory sensitivity is bounded by linear scaling with FlashAttention.*

Queueing Characteristics. Building on the per-operator compute sensitivities, we analyze how operators respond to increasing request load (RPS) using M/M/c queueing theory. Each operator is modeled as a multi-replica queueing system, where the service rate is $\mu = 1/(\text{op\_latency} \times \text{num\_layers})$, derived from the measured GPU execution times, and the arrival rate is $\lambda = \text{requests\_per\_second/batch\_size}$. Using the Erlang-C formula, we estimate waiting times and determine the minimum number of replicas required to maintain system stability under varying RPS.

Figure 6 shows heterogeneous queueing sensitivities across operators that closely reflect their compute characteristics. For attention operators, especially at longer sequence lengths, the number of replicas required grows sharply with increasing RPS, reflecting their high per-token computational cost. In MoE models like Mixtral, the FusedMoE linear operator dominates compute at short sequence lengths, leading to a

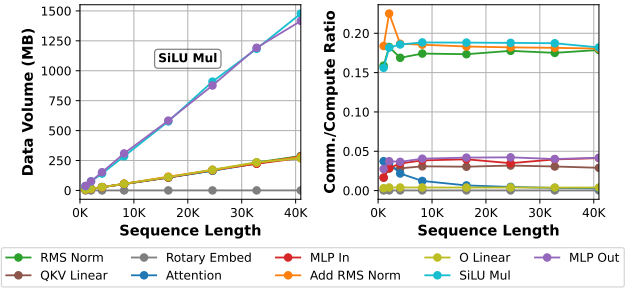


Figure 7: Operator input data volume for Qwen2-7B.

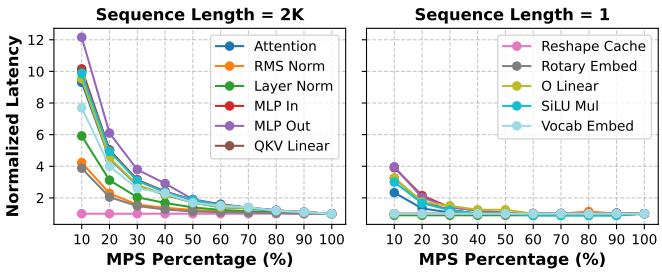
pronounced scaling of replicas even for relatively small sequences. In contrast, lighter operators, such as layer norms and embeddings, exhibit moderate sensitivity to RPS. Queueing delays increase non-linearly when replication is insufficient, so small reductions in replicas can cause disproportionately high waiting times. These observations highlight that combining compute profiling and queueing modeling enables precise, operator-specific replication strategies by selectively replicating high-demanding operators (*e.g.*, attention) while avoiding overprovisioning lightweight ones, thereby meeting end-to-end latency and throughput targets efficiently.

> **Insight 3:** *Operators exhibit diverse queueing sensitivity with increasing load. Insufficient replication causes non-linear queueing delays, emphasizing the need for operator-specific replication strategies.*

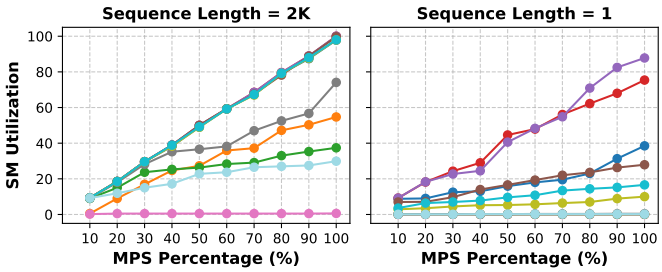
Dataflow Characteristics. We analyze data flows by quantifying the communication payload between adjacent operators in the model graph. Specifically, we perform transient memory profiling to capture the input-output scaling with sequence length and batch size for each operator, revealing a linear growth of data volume (as shown in Figure 7). Communication overhead is estimated from transfer latency, which scales proportionally with data volume. Attention and linear operators exhibit near-constant per-request data volume, whereas attention and linear operators scale with sequence length. Comparative analysis of compute versus NVLink transfer time shows transfer overhead reaching 20% for certain operators (*e.g.*, `SiLu Mul`) but remaining below 5% for most.

> **Insight 4:** *Data volume scales linearly or remains flat with sequence length across operators. Transfer overhead can reach 20% of compute time, making transfer costs non-trivial when placing operators across devices.*

Sensitivity to SM Allocation. Placing operators on GPUs, especially when multiple workloads share the same GPU, requires a precise understanding of how Streaming Multiprocessor (SM) allocation affects operator performance. This is similar to our compute characteristics study. Figure 8 illustrates how different operators respond to changes in SM allocation, which is controlled by the MPS (Multi-Process Ser-



(a) Operator normalized latency.



(b) Operator SM utilization (%).

Figure 8: Performance and SM utilization characteristics across operators (Qwen2 [20]) under varying SM allocations.

vice) percentage [31]. This analysis compares a long sequence length of 2K (prefill phase) and a short sequence length of 1 (decode phase).

For the prefill phase (2K sequence length), the top-left plot shows that as the MPS percentage increases, the normalized latency for all operators decreases significantly. We note that compute-intensive operators like Attention and MLP dominate latency, especially at lower MPS values. This is because these operators saturate the SM utilization at limited resources, and therefore, a reduction in SM allocation directly increases their latency, explaining the steep performance curve.

In the decode phase (sequence length of 1), a different pattern emerges. As shown in the top-right plot, the normalized latency for most operators remains low and relatively flat, with only a minor decrease as MPS increases. The bottom-right plot clarifies the reason: the SM utilization for these operators is low, and they do not saturate the available resources. Therefore, reducing the MPS percentage has a minimal impact on their latency. This makes a lower MPS percentage suitable for short sequences, as it allows for better resource sharing without significantly compromising performance.

Insight 5: *Operator sensitivity to SM allocation varies widely across operators, sequence lengths, and prefill/decode phases, correlating with SM utilization patterns.*

4 A Theoretical Framework for Autoscaling

Building on our systematic characterization and insights of operator performance and resource behavior, we now turn to a theoretical analysis of the benefits and the design of our operator-level model provisioning framework (Figure 9). Specifically, we decompose the problem into two key stages: (1) operator autoscaling, and (2) operator placement, which mirrors autoscaling the whole model as the scaling unit and placing each model replica to fixed devices.

In this section, we first present a theoretical formulation for operator-level autoscaling in a model inference graph. We use queueing theory to mathematically verify our insights and identify conditions broadly when operator-level autoscaling provides benefits. At a high level, a computation graph consists of operators connected via data dependencies, with each operator characterized by computation time, memory consumption, and communication cost. Given a stream of requests with a certain arrival rate (QPS) and request sequence (input) lengths, the goal for autoscaling is to scale each operator in terms of parallelism and replication to meet SLOs while minimizing total GPU usage. In parallel, the goal for operator placement is to assign scaled operators to physical devices to minimize provisioning cost (*i.e.*, devices) without SLO violations by modeling the spatial-temporal GPU utilization and capacity at the operator granularity.

4.1 Problem Formulation

Consider a directed acyclic graph (DAG) of operators $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of operators and $\mathcal{E}$ represents data dependencies. Let the input request stream $x \in X$ have an arrival rate λ (requests per second) and a request sequence length distribution $L(x)$. Each pass of the DAG corresponds to one *iteration*. For autoregressive models, a request goes through multiple iterations [38]: (1) the first iteration is ***prefill***, which processes the full input sequence (*i.e.*, length = input length); (2) subsequent ***decode*** iterations (*i.e.* length = 1) where each iteration generates an output token.

Operator Attributes. For each operator $v \in \mathcal{V}$:

- *Computation time:* $T_v = f_v(P_v, L, B)$. The computation time of each operator depends on the model parallelism P_v, request sequence length L, and batch size B.
- *Memory consumption:* $M_v = M_v^{\text{weight}} + M_v^{\text{transient}}$. Memory consumption of each operator consists of weight memory and transient memory (*e.g.*, activation), which depends on the request sequence length L and batch size B.
- *Communication time:* $C_v = u_v(P_v, L, B)$ is the communication time of operator v to its downstream operators, varying to parallelism P_v, sequence length L, and batch size B.

Queueing Model. Each operator is modeled as an $M/M/R_v$ queue, where R_v is the number of replicas. The service rate of each operator is referred to as $\mu_v = 1/T_v$. In queueing theory,

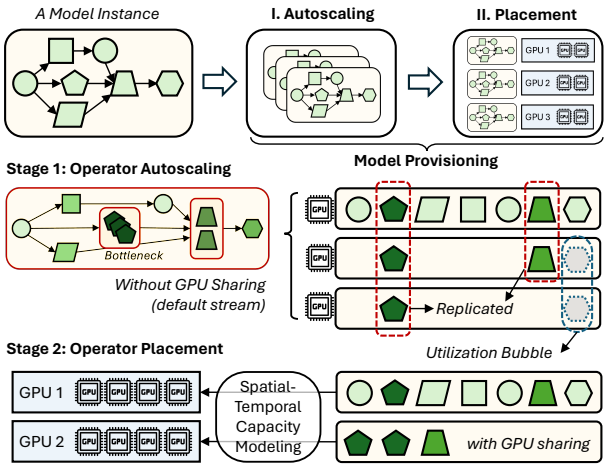


Figure 9: Model provisioning consists of (1) autoscaling and (2) placement at the operator granularity.

the expected waiting time W_v at the operator level is:

$$W_v = \frac{C(R_v, \rho_v)}{R_v \mu_v - \lambda} \quad \text{with} \quad \rho_v = \frac{\lambda}{R_v \mu_v}, \tag{1}$$

where $C(R_v, \rho_v)$ is the Erlang-C formula:

$$C(R_v, \rho_v) = \frac{(R_v \rho_v)^{R_v}}{R_v!\,(1-\rho_v)} \Big/ \sum_{k=0}^{R_v-1} \frac{(R_v \rho_v)^k}{k!} + \frac{(R_v \rho_v)^{R_v}}{R_v!\,(1-\rho_v)}. \tag{2}$$

Iteration Latency. The latency for a single DAG iteration (*i.e.*, one prefill iteration or one decode iteration) is

$$T_{\text{total}} = \sum_{v \in \text{critical path}} (T_v + W_v + C_v). \tag{3}$$

Configurations. Operator configurations are modeled as decision variables that include parallelism degree P_v, replicas R_v, and batch sizes B_v. In addition, we consider the operator-to-device assignment A_v, which impacts both communication overhead and memory feasibility. Another configuration dimension is the MPS share [31], which enables GPU sharing among operator replicas by specifying the fraction of allocated SM cores.

$$P_v \in \mathcal{P}_v = \{1, 2, 4, 8, \ldots\}, \quad \forall v \in \mathcal{V} \tag{4}$$

$$R_v, B_v \in \mathbb{Z}^+, \quad \forall v \in \mathcal{V} \tag{5}$$

$$M_v \in \mathbb{Z}, \quad M_v \in [1, 100], \quad \forall v \in \mathcal{V} \tag{6}$$

Constraints and Objectives. TTFT and TBT SLOs on the inference latency are modeled as constraints, requiring that the prefill iteration latency is below TTFT SLO while the decode iteration latency meets TBT SLO, *i.e.*, $T_{\text{total}} \leq T_{\text{SLO}}$. In addition, for each device $d \in \mathcal{D}$, the aggregate memory consumption of all operators assigned to d cannot exceed its memory capacity M_d^{cap}:

$$\sum_{v \in \mathcal{V}: A_v = d} M_v \leq M_d^{\text{cap}}, \quad \forall d \in \mathcal{D}. \tag{7}$$

Subject to this constraint, the objective is to minimize the aggregate GPU usage across all operators:

$$\min \sum_{v \in \mathcal{V}} P_v \cdot R_v, \tag{8}$$

where P_v is the degree of parallelism and R_v is the number of replicas for operator v.

4.2 Resource Management Optimization

The optimization problem formulation (Section 4.1) determines the best configurations for each operator v to minimize GPU usage while satisfying latency SLOs, taking into account computation, memory, communication, and queueing delays. However, solving such an optimization problem exactly at the operator level is computationally expensive. The configuration search space grows rapidly with the number of operators and their parameters (*e.g.*, replicas, parallelism choices), making optimal solutions impractical to obtain at fine time granularity (*e.g.*, every ten seconds). To address this, we next present algorithms that approximate the optimal solution at significantly lower overhead by decoupling autoscaling from operator placement with device sharing.

4.2.1 Operator-level Autoscaling

At runtime, each operator's throughput and latency are primarily determined by how it is parallelized, replicated, and batched. In the baseline model-level parallelism configuration [51], tensor parallelism distributes model shards across devices within a server (constrained by memory capacity and model size), while pipeline parallelism connects stages across servers. Each operator inherits this initial parallel structure from the model's deployment plan, which defines its starting parallelism degree P_v. Building on this baseline, the operator autoscaler (Stage 1 in Figure 9) dynamically adjusts P_v, R_v, B_v for each operator to (1) satisfy latency SLOs on TTFT or TBT, and (2) greedily minimize resource usage ($\sum_v P_v \cdot R_v$). We refer to the full algorithm in Algorithm 1.

First, for each operator, we scan $b \in \{1, \ldots, B_v^{\max}\}$ with an initial P_v inherited from model-level parallelism, set $R_v(b) \leftarrow \lceil \lambda_v / \mu_v(b, p_v) \rceil$, and select the (B_v, R_v) that minimizes the total sojourn time $\sum S_v = T_v(P_v, b) + W_v(\lambda_v, R_v(b), T_v)$ while maintaining stability [4, 12]. This gives a set of low-latency, stable per-operator configurations that seed the global greedy search.

The global search proceeds iteratively. We repeatedly evaluate the current iteration latency T_{total} along the critical path. If $T_{total} > SLO$, we *upscale* at the current bottleneck operator

v (the operator on the critical path with the largest S_v). Upscaling prefers the smallest change that most effectively reduces T_{total}: increasing R_v by one, optionally co-tuning (B_v, P_v) to exploit batching or parallelism efficiency improvements. After each candidate move, we recompute λ, update all affected sojourn times S_v, and re-evaluate T_{total}, accepting the move that maximally reduces T_{total} (or minimally increases resource usage while restoring $T_{total} \leq SLO$).

If $T_{total} < SLO$ (by more than a tolerance buffer ε), we attempt to release resources by downscaling the bottleneck operator v on the critical path. The candidate moves include: decrease R_v by one (if stable) and optionally adjust (B_v, P_v) to compensate. Among feasible moves that keep $T_{total} \leq SLO$, we select the one with the best objective (*e.g.*, largest reduction in $\sum_v P_v \cdot R_v$ or total compute cost). The loop terminates when no local move can improve the objective without violating the SLO (within ε), or when T_{total} cannot be restored to the SLO via upscaling (*i.e.*, infeasible SLOs).

The search space is discrete and finite: $R_v \in \mathbb{N}_+$, $B_v \in \{1, \ldots, B_v^{\max}\}$, $P_v \in \mathcal{P}_v$. Each greedy step only considers local changes at the bottleneck, which concentrates optimization effort along the critical path where it most affects the iteration latency. Since batch size influences both service rate and queuing, we always recompute arrival rates and sojourn times after every accepted move. While not globally optimal, this greedy algorithm converges quickly to SLO-feasible, resource-efficient configurations and is lightweight enough to run online as workload traffic evolves.

Algorithm 1 Greedy Operator-Level Autoscaling

Require: DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, QPS q, batch limits $B_i^{\max}$, parallelism sets $\mathcal{P}_v$, latency functions $T_v(b,p)$ from profiling, SLO T_{slo} with a buffer ε

Initialize $p_v \leftarrow \min \mathcal{P}_v$ and $b_v \leftarrow 1$ for all $v \in \mathcal{V}$
for all $v \in \mathcal{V}$ **do** ▷ Per-operator initialization
  $\lambda \leftarrow \textsc{ArrivalRates}(\mathcal{G}, q, b_v)$
  $r_v \leftarrow \lceil \lambda / \mu_v(1, p_v) \rceil$ where $\mu_v(b,p) = b/T_v(b,p)$
  $(b_v, r_v) \leftarrow \arg \min_{b \in \{1,\ldots,B_v^{\max}\}} s_v(\lambda, \lceil \lambda/\mu_v(b,p_v) \rceil, b, p_v)$
end for
Recompute $\lambda \leftarrow \textsc{ArrivalRates}(\mathcal{G}, q, \{b_v\})$
$s_v \leftarrow W_v(\lambda_v, r_v, \mu_v(b_v,p_v)) + T_v(b_v,p_v)/b_v$ for all v
$T \leftarrow \textsc{CriticalPathLatency}(G, \{s_v\})$
while true **do**
  if $T \leq T_{slo} - \varepsilon$ **then** ▷ Scaling down
    $j \leftarrow \textsc{BottleneckOnCriticalPath}(G, \{s_v\})$
    $\mathcal{M} \leftarrow \{(r_j - 1, b_j, p_j)\} \cup \{(r_j - 1, b, p_j) \mid b \in [b_j, B_j^{\max}]\} \cup \{(r_j - 1, b, p) \mid b \in [b_j, B_j^{\max}], p \in \mathcal{P}_j\}$
    Filter $\mathcal{M}$ with stability check $\lambda_j < (r'_j)\mu_j(b', p')$
    For each $m \in \mathcal{M}$: tentatively recompute T'
    Choose $m^\star \in \arg\min\{\textsc{Cost}(\mathbf{r}', \mathbf{p}') \mid T' \leq T_{slo}\}$
    if $m^\star$ exists **then** apply $m^\star$, set $T \leftarrow T'$, continue
    else break ▷ No further safe downscale
    end if
  else if $T > T_{slo}$ **then** ▷ Scaling up
    $j \leftarrow \textsc{BottleneckOnCriticalPath}(G, \{s_v\})$
    $\mathcal{M} \leftarrow \{(r_j + 1, b_j, p_j)\} \cup \{(r_j + 1, b, p_j) \mid b \in [1, B_j^{\max}]\} \cup \{(r_j + 1, b, p) \mid b \in [1, B_j^{\max}], p \in \mathcal{P}_j\}$
    For each $m \in \mathcal{M}$: tentatively re-evaluate T'
    Choose $m^\star \in \arg\max\{T - T'\}$; Prefer the smallest Δr_j that achieves $T' \leq T_{slo}$)
    if $m^\star$ exists **then** apply $m^\star$, set $T \leftarrow T'$, continue
    else break ▷ Cannot improve further
    end if
  else break ▷ Within tolerance of SLO
  end if
end while
return $\{(r_i, b_i, p_i)\}_{i \in V}$

4.2.2 Operator-to-Device Placement

Algorithm 2 presents the algorithm that maps operator replicas to devices while minimizing device usage under memory and SLO constraints. We first compute the baseline $k_{\text{base}} = \min_v r_v$ and deploy those full model instances to form $\mathcal{D}_{\text{base}}$. Extra replicas $\mathcal{R}_{\text{extra}}$ are sorted by T_v (largest first) and placed greedily: for each (v,k) we probe devices in $\mathcal{D}_{\text{base}}$, reject any that violate memory M_d or where the interference-adjusted latency $T'_v = T_v \cdot I_{d,v}(b_v, p_v)$ would make the recomputed end-to-end latency exceed the SLO, and score feasible candidates by weighted residual slack (memory and compute) choosing the best; if none fit, a new device is provisioned. As illustrated in Figure 9 (Stage 2), the extra scaled-out replicas are preferentially colocated with existing model instances, and only provisioned on new devices when memory or compute capacity is insufficient.

Default Stream Constraint. To this end, we assume multi-stream is enabled on GPUs that support time or space sharing. However, in older GPUs that do not support GPU sharing, replicas of the same operators are executed sequentially as a single stream in the default stream setup. As a result, scaled-out replicas are placed on separate devices, eliminating the possibility of reducing the number of active GPUs compared to model-level autoscaling. While this constraint removes opportunities for device savings, it opens the door for energy optimizations: lower compute density on each scaled-out device can reduce overall energy consumption. To capture this effect, we introduce an operator-level energy attribution model for each request:

$$E_v = \alpha_v \cdot P_v \cdot R_v \cdot (W_v + T_v) + \beta_v \cdot T_v, \tag{9}$$

where α_v and β_v are power coefficients for device usage (idle power) and active computation (dynamic power).

Algorithm 2 Greedy Operator-Level Placement

Require: DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, config $\{(r_v, b_v, p_v)\}_{v \in \mathcal{V}}$ from Alg. 1, device set $\mathcal{D}$, device capacities $\{M_d, U_d\}_{d \in \mathcal{D}}$, interference model $I_{d,v}(b,p) \geq 1$ (from profiling).

$k_{\text{base}} \leftarrow \min_{v \in \mathcal{V}} r_v$ ▷ Number of full model instances
Construct replica sets:
  $\mathcal{R}_{\text{base}} \leftarrow \{(v,i) \mid v \in \mathcal{V},\ i \in [1, k_{\text{base}}]\}$
  $\mathcal{R}_{\text{extra}} \leftarrow \{(v,i) \mid v \in \mathcal{V},\ i \in [k_{\text{base}}+1, r_v]\}$
Sort $(v,k) \in \mathcal{R}_{\text{extra}}$ in descending order of T_v
$\mathcal{D}_{\text{base}} \leftarrow \text{DEPLOYMODELINSTANCE}(\mathcal{R}_{\text{base}})$
$\mathcal{D}_{\text{empty}} \leftarrow \mathcal{D} \setminus \mathcal{D}_{\text{base}}$
for all $(v,k) \in \mathcal{R}_{\text{extra}}$ **do**
  $Candidates \leftarrow \emptyset$
  for all $d \in \mathcal{D}_{\text{base}}$ **do** ▷ Try existing devices first
    if $MemLoad_d + m_v > M_d$ **then**
      continue
    end if
    $T'_v \leftarrow T_v \cdot I_{d,v}(b_v, p_v)$
    if $\text{RECOMPUTELATENCY}(\mathcal{G}) > SLO$ **then**
      continue
    end if
    $slack\_mem \leftarrow M_d - (MemLoad_d + m_v)$
    $slack\_comp \leftarrow U_d - (CompLoad_d + T'_v)$
    $Candidates.\text{APPEND}(d)$
  end for
  if $Candidates == \emptyset$ **then** ▷ No existing device fits
    $d_{\text{new}} \leftarrow \text{PROVISIONDEVICE}(\mathcal{D}_{\text{empty}}, \mathcal{D}_{\text{base}})$
    $d^\star \leftarrow d_{\text{new}}$
  else
    $\text{COMPUTEWEIGHTEDSLACK}(Candidates)$
    $d^\star \leftarrow \arg\max_{(d) \in Candidates} slack$
  end if
  $Placement \leftarrow \text{ASSIGNOPERATOR}(v,k,d^\star)$
end for
return *Placement*

4.2.3 Baselines

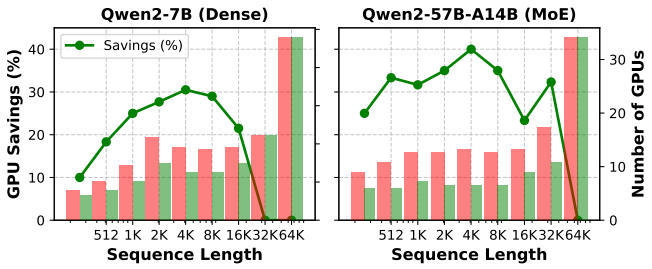
Model-level Autoscaling and Provisioning. As a comparison, model-level autoscaling treats the entire model as a monolithic unit, enforcing uniformity across all operators. Specifically, all operators share the same batch size B and the same number of replicas R, with parallelism P fixed by the chosen tensor/pipeline partitioning strategy. Rather than tuning per-operator parameters, the autoscaler adjusts (B,R) globally to meet latency SLOs on TTFT or TBT. Model-level autoscaling therefore provides a coarse-grained but stable baseline: it captures system-wide scaling trends but lacks the flexibility to exploit per-operator heterogeneity in workload intensity or compute efficiency, limiting opportunities for fine-grained resource optimization compared to operator-level autoscaling. Every scaled-out model replica is placed onto a new set of GPU devices without sharing.

Brute-force Approach. As an oracle baseline, brute-force search enumerates all operator configurations (P_v, R_v, B_v), evaluates end-to-end latency, and selects the resource-minimal SLO-feasible point. While this guarantees optimality, the combinatorial space $O\left(\prod_v |\mathcal{P}_v| \cdot B_v^{\max} \cdot R_v^{\max}\right)$ makes it computationally prohibitive for online use.

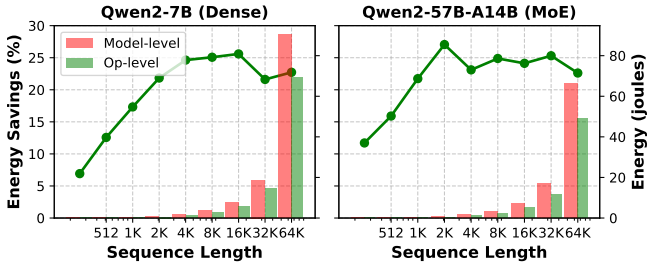
4.3 Experimental Analysis

We now evaluate the effectiveness of operator-level autoscaling and placement compared to conventional model-level provisioning. Our goal is to quantify the resource and efficiency gains (*i.e.*, in device usage, energy consumption, and memory utilization) while preserving latency SLOs. Using the theoretical formulation and algorithms described in the previous section, we simulate autoscaling behavior without executing full model inference [27]. We analyze how operator-level scaling adapts under varying workload and SLO conditions, including changes in sequence length, batch size, and request arrival rate (QPS). Experiments are conducted using representative models, Qwen2-7B [20] and Qwen2-MoE [19], which capture both dense and sparse (MoE) inference characteristics. To simulate real-world workloads, we adopt production LLM inference traces from Azure [48] and Moonshot AI [42]. Through these experiments, we highlight when and why fine-grained operator-level autoscaling yields substantial savings over traditional model-level scaling, especially under heterogeneous or dynamic workloads.

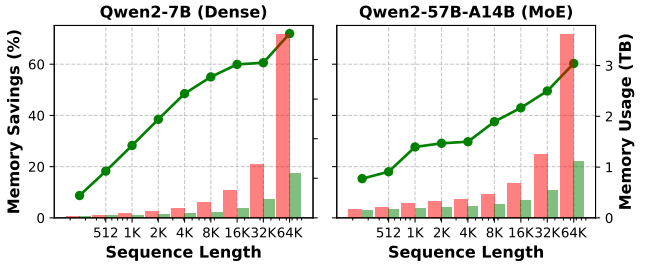
Varying Request Sequence Length. We first examine how varying the input sequence length affects autoscaling behavior and resource allocation under SLOs, capturing the distinct scaling patterns between prefill- and decode-dominated workloads. As shown in Figure 10, increasing sequence length yields varying savings across GPU devices and energy, and memory resources, with diminishing returns beyond 8K tokens. GPU savings (Figure 10a) peak around 30% at 4K tokens for the dense model and 40% for MoE as operator-level provisioning effectively consolidates workloads during longer prefill phases, but drop at very long sequences where SM saturation limits further gains from GPU sharing. MoE models benefit from higher savings because of their sparsity and more diversity in operator sensitivity. Energy savings (Figure 10b) follow a similar trend, reaching up to 25% reduction at peak, mainly due to the savings in devices. When there is no savings in the GPU devices, the energy consumption for operator-level provisioning is low due to only scaling out bottlenecked operators. Memory savings (Figure 10c) grow more steadily with sequence length, surpassing 60% at 32K tokens for Qwen2-7B and 64K tokens for Qwen-MoE. Unlike GPU and energy savings, this trend arises not from operator colocation but from selective scaling: only compute-bottlenecked operators are scaled out, yielding substantial memory reductions from the non-sensitive operators, compared to uniformly scaling all operators in model-level provisioning.



(a) GPU savings.

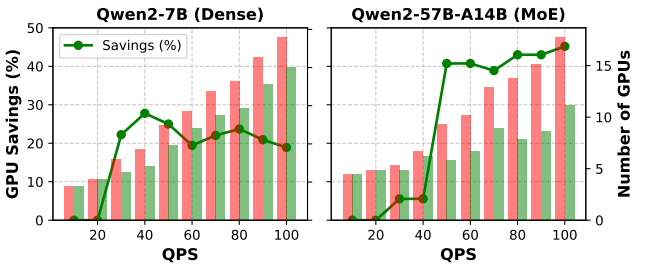


(b) Energy savings.

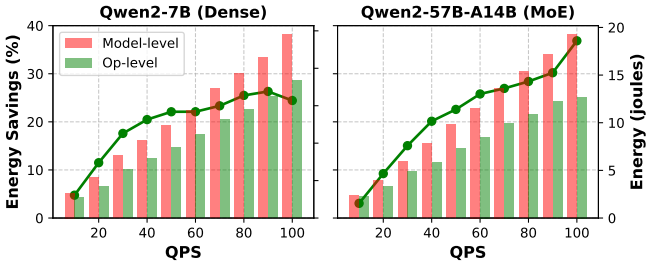


(c) Memory savings.

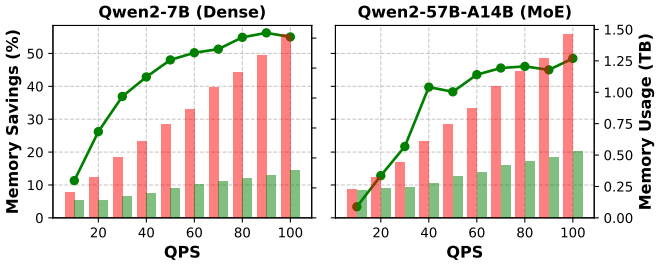
Figure 10: Benefits of operator-level resource management under varying sequence lengths.



(a) GPU savings.



(b) Energy savings.



(c) Memory savings.

Figure 11: Benefits of operator-level resource management under varying QPS.

Insight 6: *Operator-level autoscaling yields the largest benefits at moderate sequence lengths, before SM contention tightens SLO margins and reduces flexibility at very long contexts. Even so, it achieves up to 25% energy and 60% memory savings across workloads.*

Varying QPS. Next, we analyze the impact of request arrival rate (QPS) on operator-level versus model-level autoscaling, highlighting how operator-level granularity enables more resource-efficient scaling decisions to mitigate queueing delays. As shown in Figure 11, operator-level autoscaling consistently delivers higher resource savings across GPU, energy, and memory compared to model-level provisioning, especially at moderate QPS. For a dense model Qwen2-7B, GPU savings (Figure 11a) peak around 30% near 40 QPS, where operator-level scaling consolidates workloads and avoids overprovisioning during bursts. Beyond this point, savings fluctuate slightly as SM saturation and tighter latency margins reduce opportunities for GPU sharing. Energy savings (Figure 11b) exhibit a similar pattern, reaching up to 25% at high QPS for Qwen2-7B, primarily driven by reduced device counts and selective operator scaling rather than uniform expansion. Memory savings (Figure 11c) grow steadily with QPS, surpassing 50% at 100 QPS, reflecting a similar trend with sequence length growth.

The Qwen-MoE models exhibit a similar scaling behavior in terms of resource savings. For both models, GPU savings remain negligible at low QPS (<20), where the provisioned model instance can handle limited traffic demands without scaling under the SLOs.

Insight 7: *Operator-level autoscaling achieves its largest benefits under moderate to high QPS, where fine-grained scaling mitigates queueing delays without excessive resource inflation, resembling the sequence length trend.*

Prefill vs. Decode. We compare operator-level and model-

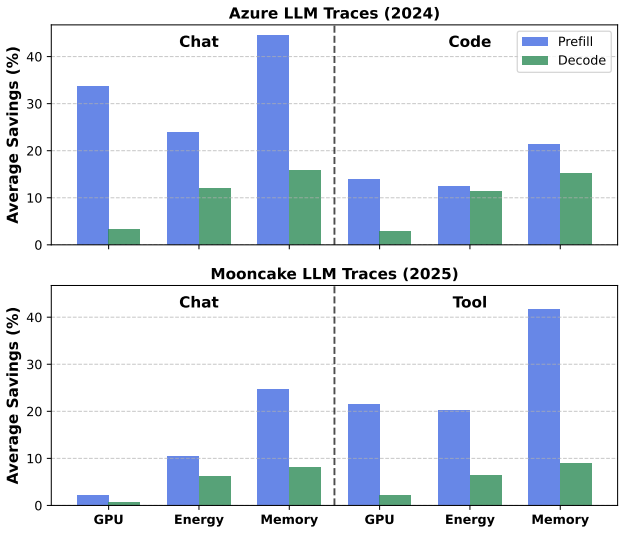


Figure 12: Benefits of operator-level resource management in prefill vs. decode stages for Qwen2-7B.

level provisioning behavior across the prefill (known to be compute-bound [38]) and decode stages (memory-bound), illustrating how their distinct computational and temporal characteristics drive different scaling needs. We analyze two production-scale LLM inference traces from (1) Azure LLM inference cluster [48] and (2) Mooncake LLM serving platform [42]. As shown in Figure 12, operator-level resource management yields consistently higher savings during the prefill stage across all workloads. On Azure traces, prefill achieves up to 35% GPU, 25% energy, and 45% memory savings for chat services, while decode savings remain modest (below 15%). Savings in coding services are less due to its low QPS in the traces. Similarly, in Mooncake traces, prefill achieves up to 22% GPU savings, 20% energy, and 41% memory, higher than the savings achieved during decode. These results highlight that prefill stages benefit more from operator-level provisioning and scaling due to its denser compute utilization and shorter execution bursts.

Insight 8: *Prefill stages offer substantially higher optimization potential than decode—up to 2–3× greater resource savings, as they are more compute-intensive and bursty, making them ideal targets for fine-grained operator-level model provisioning and scaling.*

Large vs. Small Models. We analyze how model size influences autoscaling effectiveness using the Qwen2 family, assuming all models are served under the same SLO target to isolate the effect of scale. As shown in Figure 13, operator-level provisioning yields substantial gains across all model sizes, with increasing benefits as model scale grows. For smaller models like Qwen2-0.5B, savings are modest (below 15%) since coarse-grained model-level provisioning is

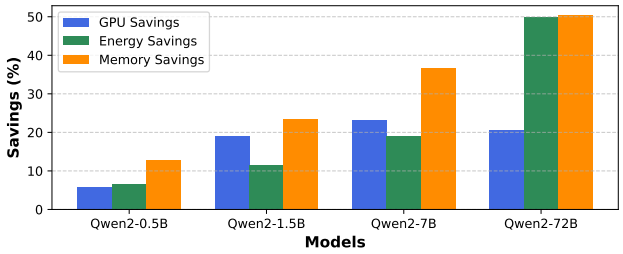


Figure 13: Benefits of operator-level resource management across various-sized models

already sufficient to meet SLOs. As model size increases to Qwen2-1.5B and Qwen2-7B, GPU and energy savings rise to around 20–30%, while memory savings exceed 35%, driven by more diverse operator-level utilization patterns that can be effectively co-scheduled. For the largest model, Qwen2-72B, both energy and memory savings peak near 50%, highlighting that large models benefit most from operator-level provisioning, which dynamically reallocates resources to mitigate fragmentation and idle GPU spaces. If the SLO were instead proportional to model size, we would expect the relative savings to remain similar across models, as larger models effectively replicate the same layer structure, leading to comparable operator-level efficiency patterns.

Insight 9: *Larger models amplify the benefits of operator-level autoscaling under a fixed SLO, while proportional SLO scaling yields similar relative savings since operator-level scaling patterns remain consistent across sizes.*

How Far from the Oracle? We quantify the optimality gap between the proposed operator-level autoscaling algorithm (Section 4.2) and the brute-force oracle. Across workload conditions with varying QPS at a sequence length of 1K, the average resource savings gap is 8%, suggesting that the greedy algorithm attains most of the theoretical optimum with significantly lower computational overhead.

5 Discussion

When to Adopt Operator-level Autoscaling? Operator-level autoscaling is most beneficial when workloads are latency-tolerant and dominated by heterogeneous operator behaviors. In contrast, inference systems pursuing ultra-low latency through megakernel fusion [3] (where all operators are fused into a single large kernel) leave limited room for operator-level provisioning and scaling. Thus, operator-level autoscaling strikes a balance between efficiency and flexibility, favoring modular deployments and scaling over monolithic kernels.

Prefill-Decode Disaggregation or Co-location? Our analysis (Section 4.3) shows that prefill stages, being compute-intensive and variable in demand, gain the most from fine-

grained autoscaling. Decode, while often constrained by tighter latency budgets, benefits less from operator-level provisioning and scaling. This suggests that enabling prefill–decode disaggregation complements operator-level scaling by exposing varying extents of optimization opportunities across stages. In prefill-decode co-location cases with chunked prefill, the maximum sequence length is effectively bounded by the chunk size, altering operator load distribution. Operator-level autoscaling remains valuable in such cases, as it can adapt provisioning to dynamic chunking [2].

Integration with Inference Schedulers. Operator provisioning and scaling should be co-optimized with request scheduling at the instance and cluster level to minimize inter-operator communication and resource fragmentation. Integrating scaling logic with LLM inference schedulers allows coordinated resource allocation that respects both topology and operator affinity, which we leave to future work.

6 Related Work

LLM Serving Optimization. Prior work on optimized LLM serving, including batch scheduling [40, 45, 49, 57], chunked prefill [1], KV-cache management [13, 23, 42], and energy or power management [37, 47, 48], are complementary to our contribution, as they target the serving engine and scheduler layers, whereas we focus on the *model execution phase* through operator-level autoscaling and placement.

In addition, there has been a trend of disaggregation on generative model serving: (1) Prefill-Decode disaggregation [38, 59] separates the two phases of generation, enabling independent autoscaling. (2) Encoder–LLM disaggregation [8, 43, 46] decouples multimodal encoders from LLM backends, allowing modality-specific scaling. (3) VAE-DiT disaggregation [15] enables specialized autoscaling in video/image generation pipelines. (4) MA parallelism [6, 28, 52, 61, 62] disaggregates attention and MLP/MoE layers for independent deployment to achieve higher throughput. From this perspective, our work pushes disaggregation to a finer granularity: the *operator level*. We examine operator heterogeneity and the benefits of conceptually decoupling operators to enable fine-grained autoscaling and SM-aware placement, without necessarily disaggregating them across separate devices.

Autoscaling Policies. As mentioned in Section 2.2, AIBrix [50], DynamoLLM [48], Chiron [39], and DeepServe [14] propose model-level autoscaling *policies* that combine demand prediction with replica management. Our work is orthogonal: rather than policy design, we contribute a new autoscaling *mechanism* at the operator level, which benefit from advanced autoscaling policies in this field.

Enabling Multi-Stream for Efficiency. Recent work [22, 26, 60] has leveraged multi-stream processing on GPUs to improve utilization and throughput. For example, NanoFlow [60] exploits intra-device parallelism by overlapping computation with I/O, thereby increasing LLM serving throughput. Similarly, Pod-Attention [22] collocates prefill and decode stages to jointly utilize compute and memory bandwidth.

Operator-level Optimization. Several efforts have explored operator-level optimizations from complementary perspectives, distinct from our focus on autoscaling and fine-grained resource management. For instance, μ-Serve [44] applies operator-level GPU frequency scaling for power efficiency; MegaKernel [53] fuses all operators into a single kernel to minimize latency; operator fusion techniques [24] reduce launch overhead; and customized attention kernels such as FlashInfer [56] optimize critical operators for lower latency.

Multi-model Multiplexing. In GPU clusters that serve multiple LLMs, prior work has explored spatial and temporal multiplexing to improve cluster utilization and reduce serving costs [9, 25, 41, 54, 55, 58]. While this paper focuses on single-model serving, its design naturally extends to multi-model scenarios. By independently managing the scaling and provisioning of each model, it enables faster switching across models and more flexible spatial sharing of limited GPU resources.

7 Conclusion

In this paper, we present a systematic characterization and theoretical framework for operator-level autoscaling and placement in large-model inference systems. Through queueing-based modeling, we decompose inference provisioning into fine-grained operator units, each governed by computation, memory, and communication tradeoffs. Our analysis demonstrates that, compared to traditional model-level scaling, operator-level resource management substantially improves GPU, memory, and energy efficiency under varying sequence lengths, request arrival rates, and model sizes—achieving up to 40% GPU and 35% energy savings while preserving SLOs. These results highlight the potential of shifting from coarse, model-centric scaling to a fine-grained, ***operator-centric inference architecture***.

Looking ahead, our findings point to a future inference system design that exploits operator-level provisioning in the model runtime layer. With continuous workload monitoring or prediction, heterogeneous operators can be co-scheduled across shared devices, and exploit spatial-temporal utilization patterns to minimize cost and energy while preserving latency SLOs. This fine-grained, operator-aware resource management offers a foundation for the next generation of elastic and efficient large-scale inference infrastructures.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming

Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[2] Amey Agrawal, Haoran Qiu, Junda Chen, Íñigo Goiri, Chaojie Zhang, Rayyan Shahid, Ramachandran Ramjee, Alexey Tumanov, and Esha Choukse. Medha: Efficiently serving multi-million context length LLM inference requests without approximations. *arXiv preprint arXiv:2409.17264*, 2024.

[3] Benjamin Spector, Jordan Juravsky, Stuart Sul, Owen Dugan, Dylan Lim, Dan Fu, Simran Arora, Chris Ré. Look Ma, No Bubbles! Designing a Low-Latency Megakernel for Llama-1B. Accessed from https://hazyresearch.stanford.edu/blog/2025-05-27-no-bubbles, 2025.

[4] Cheng-Shang Chang. Stability, queue length, and delay of deterministic and stochastic queueing networks. *IEEE Transactions on Automatic Control*, 39(5):913–931, 1994.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of The 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with Pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[7] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems (NeurIPS 22)*, 35:16344–16359, 2022.

[8] Xianzhe Dong, Tongxuan Liu, Yuting Zeng, Liangyu Liu, Yang Liu, Siyu Wu, Yu Wu, Hailong Yang, Ke Zhang, and Jing Li. HydraInfer: Hybrid disaggregated scheduling for multimodal large language model serving. *arXiv preprint arXiv:2505.12658*, 2025.

[9] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. MuxServe: Flexible spatial-temporal multiplexing for multiple LLM serving. *arXiv preprint arXiv:2404.02015*, 2024.

[10] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM:low-latency serverless inference for large language models. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 135–153, 2024.

[11] GitHub. vLLM Production Stack. https://github.com/vllm-project/production-stack, 2025.

[12] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[13] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. MemServe: Context caching for disaggregated LLM serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, 2024.

[14] Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Jie Meng, Baoquan Zhang, Shining Wan, et al. DEEPSERVE: Serverless large language model serving at scale. In *Proceedings of the 2025 USENIX Annual Technical Conference (ATC)*, 2024.

[15] Heyang Huang, Cunchen Hu, Jiaqi Zhu, Ziyuan Gao, Liangliang Xu, Yizhou Shan, Yungang Bao, Sun Ninghui, Tianwei Zhang, and Sa Wang. DDiT: Dynamic resource allocation for diffusion transformer model serving. *arXiv preprint arXiv:2506.13497*, 2025.

[16] Tao Huang, Pengfei Chen, Kyoka Gong, Jocky Hawk, Zachary Bright, Wenxin Xie, Kecheng Huang, and Zhi Ji. ENOVA: Autoscaling towards cost-effective and stable serverless LLM serving. *arXiv preprint arXiv:2407.09486*, 2024.

[17] Hugging Face. Llama-3-8b. https://huggingface.co/meta-llama/Meta-Llama-3-8B, 2025.

[18] Hugging Face. Mixtral-8x7B-v0.1. https://huggingface.co/mistralai/Mixtral-8x7B-v0.1, 2025.

[19] Hugging Face. Qwen2-57B-A14B. https://huggingface.co/Qwen/Qwen2-57B-A14B-Instruct, 2025.

[20] Hugging Face. QWen2-7B-Instruct. https://huggingface.co/Qwen/Qwen2-7B-Instruct, 2025.

[21] Hugging Face. Qwen2.5-VL-32B. https://huggingface.co/Qwen/Qwen2.5-VL-32B-Instruct, 2025.

[22] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. Pod-Attention: Unlocking full prefill-decode overlap for faster LLM inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 25)*, pages 897–912, 2025.

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 23)*, 2023.

[24] Jinhao Li, Jiaming Xu, Shan Huang, Yonghua Chen, Wen Li, Jun Liu, Yaoxiu Lian, Jiayi Pan, Li Ding, Hao Zhou, et al. Large language model inference acceleration: A comprehensive hardware perspective. *arXiv preprint arXiv:2410.04466*, 2024.

[25] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[26] Zejia Lin, Hongxin Xu, Guanyi Chen, Xianwei Zhang, and Yutong Lu. Bullet: Boosting GPU utilization for LLM serving via dynamic spatial-temporal orchestration. *arXiv preprint arXiv:2504.19516*, 2025.

[27] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, et al. nnScaler: Constraint-guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 347–363, 2024.

[28] Ziming Liu, Boyu Tian, Guoteng Wang, Zhen Jiang, Peng Sun, Zhenhua Han, Tian Tang, Xiaohe Hu, Yanmin Jia, Yan Zhang, et al. Expert-as-a-service: Towards efficient, scalable, and robust large-scale MoE serving. *arXiv preprint arXiv:2509.17863*, 2025.

[29] Microsoft Azure. Azure VM NDm-A100-v4 sizes series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndma100v4-series, 2024.

[30] NVIDIA. Documentation on NVIDIA Multi-Instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2025.

[31] NVIDIA. Documentation on NVIDIA Multi-Procecss Service (MPS). https://docs.nvidia.com/deploy/mps/index.html, 2025.

[32] NVIDIA. NVIDIA DCGM. https://developer.nvidia.com/dcgm, 2025.

[33] NVIDIA. NVIDIA Green Context Documentation. https://docs.nvidia.com/cuda/cuda-driver-api/group_CUDA_GREEN_CONTEXTS.html, 2025.

[34] OpenAI. Introducing ChatGPT. https://openai.com/index/chatgpt/, 2022.

[35] OpenAI. Introducing Deep Research. https://openai.com/index/introducing-deep-research/, 2025.

[36] OpenAI Platform. OpenAI Realtime API. https://platform.openai.com/docs/guides/realtime, 2025.

[37] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. Characterizing Power Management Opportunities for LLMs in the Cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 24)*, 2024.

[38] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA 24)*, pages 118–132. IEEE, 2024.

[39] Archit Patke, Dhemath Reddy, Saurabh Jha, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Hierarchical autoscaling for large language model serving with Chiron. *arXiv preprint arXiv:2501.08090*, 2025.

[40] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Queue Management for SLO-Oriented Large Language Model Serving. In *Proceedings of the 15th ACM Symposium on Cloud Computing (SoCC 24)*, 2024.

[41] Yifan Qiao, Shu Anzai, Shan Yu, Haoran Ma, Shuo Yang, Yang Wang, Miryung Kim, Yongji Wu, Yang Zhou, Jiarong Xing, Joseph E. Gonzalez, Ion Stoica, and Harry Xu. Conserve: Fine-grained gpu harvesting for llm online and offline co-serving, 2025.

[42] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. *arXiv preprint arXiv:2407.00079*, 2024.

[43] Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, et al. ModServe: Scalable and resource-efficient large multimodal model serving. *arXiv preprint arXiv:2502.00937*, 2025.

[44] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. Power-aware deep learning model serving with μ-Serve. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 75–93, 2024.

[45] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. Efficient Interactive LLM Serving with Proxy Model-based Sequence Length Prediction. In *The 5th International Workshop on Cloud Intelligence / AIOps at ASPLOS 2024*, 2024.

[46] Gursimran Singh, Xinglu Wang, Ivan Hu, Timothy Yu, Linzi Xing, Wei Jiang, Zhefeng Wang, Xiaolong Bai, Yi Li, Ying Xiong, et al. Efficiently serving large multimedia models using epd disaggregation. *arXiv preprint arXiv:2501.05460*, 2024.

[47] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Esha Choukse, Haoran Qiu, Rodrigo Fonseca, Josep Torrellas, and Ricardo Bianchini. TAPAS: Thermal-and Power-Aware Scheduling for LLM Inference in Cloud Platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1266–1281, 2025.

[48] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. DynamoLLM: Designing LLM inference clusters for performance and energy efficiency. In *Proceedings of the 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA 25)*, pages 1348–1362. IEEE, 2025.

[49] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic Scheduling for Large Language Model Serving. *arXiv preprint arXiv:2406.03243*, 2024.

[50] The AIBrix Team, Jiaxin Shan, Varun Gupta, Le Xu, Haiyang Shi, Jingyuan Zhang, Ning Wang, Linhui Xu, Rong Kang, Tongping Liu, et al. AIBrix: Towards scalable, cost-effective large language model inference infrastructure. *arXiv preprint arXiv:2504.03648*, 2025.

[51] vLLM. Distributed Inference and Serving. https://docs.vllm.ai/en/latest/serving/distributed_serving.html, 2024.

[52] Bin Wang, Bojun Wang, Changyi Wan, Guanzhe Huang, Hanpeng Hu, Haonan Jia, Hao Nie, Mingliang Li, Nuo Chen, Siyu Chen, et al. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding. *arXiv preprint arXiv:2507.19427*, 2025.

[53] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A multi-level superoptimizer for tensor programs. In *Proceedings of The 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, Boston, MA, July 2025. USENIX Association.

[54] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. Aegaeon: Effective GPU pooling for concurrent LLM serving on the market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 1030–1045, 2025.

[55] Jiarong Xing, Yifan Qiao, Simon Mo, Xingqi Cui, Gur-Eyal Sela, Yang Zhou, Joseph Gonzalez, and Ion Stoica. Towards efficient and practical gpu multitasking in the era of llm. *arXiv preprint arXiv:2508.08448*, 2025.

[56] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. FlashInfer: Efficient and customizable attention engine for LLM inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[58] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, et al. Prism: Unleashing gpu sharing for cost-efficient multi-llm serving. *arXiv preprint arXiv:2505.04021*, 2025.

[59] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang.

DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[60] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, et al. NanoFlow: Towards optimal large language model serving throughput. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 749–765, 2025.

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

[62] Pengfei Zuo, Huimin Lin, Junbo Deng, Nan Zou, Xingkun Yang, Yingyu Diao, Weifeng Gao, Ke Xu, Zhangyu Chen, Shirui Lu, et al. Serving large language models on Huawei CloudMatrix384. *arXiv preprint arXiv:2506.12708*, 2025.