JANUS: Disaggregating Attention and Experts for Scalable MoE Inference

Zhexiang Zhang[1*], Ye Wang[1*], Xiangyu Wang[2], Yumiao Zhao[3], Jingzhe Jiang[1],
Qizhen Weng[2], Shaohuai Shi[4], Yin Chen[2], Minchen Yu[1†]

[1]*The Chinese University of Hong Kong, Shenzhen*
[2]*Institute of Artificial Intelligence (TeleAI), China Telecom*
[3]*Shenzhen Loop Area Institute* [4]*Harbin Institute of Technology, Shenzhen*

Abstract

Large Mixture-of-Experts (MoE) model inference is challenging due to high resource demands and dynamic workloads. Existing solutions often deploy the entire model as a single monolithic unit, which applies a unified resource configuration to both attention and expert modules despite their different requirements, leading to limited scalability and resource inefficiency. In this paper, we propose JANUS, a scalable MoE inference system that disaggregates attention and experts on separate GPU sub-clusters, enabling each module to be managed and scaled independently. JANUS incorporates three key designs for efficient, disaggregated MoE inference. First, it proposes an adaptive two-phase communication scheme that exploits intra- and inter-node bandwidth hierarchies for low-latency data exchange. Second, motivated by the memory-bound nature of MoE modules, JANUS introduces a lightweight scheduler and implements it as a GPU kernel to balance the number of activated experts across GPUs at minimal overhead, thereby reducing inference latency. Third, JANUS performs fine-grained resource management to dynamically adjust expert placement and independently scale attention and MoE resources to improve overall efficiency. Evaluation shows JANUS achieves up to 3.9× higher per-GPU throughput than state-of-the-art systems while meeting per-token latency requirements.

1 Introduction

Recent advancements in Large Language Models (LLMs) have driven their widespread adoption across diverse domains. As mainstream LLMs scale to ever-larger parameter sizes, the Mixture-of-Experts (MoE) has emerged as a dominant architecture, which effectively expands model capacity without proportionally increasing per-token computation [2, 11, 30, 31]. Compared with traditional dense LLMs, MoE models retain the attention layers but replace each Feed-Forward Network (FFN) layer with an MoE layer composed of multiple FFNs as experts. These experts are sparsely activated at inference time—only a small subset is invoked for each token—allowing the model to deliver higher effective capacity while maintaining similar computational demands to dense LLMs.

However, serving large MoE models in real-world, dynamic environments remains challenging due to three key characteristics. First, online LLM inference services must handle *dynamic workloads*, including fluctuating request arrival rates and highly variable input and output token lengths [22, 26, 33, 35, 37], which makes it hard to consistently satisfy latency Service Level Objectives (SLOs) such as time-per-output-token (TPOT). Second, MoE models have *very large memory footprints—dominated by the expert parameters* [2, 11]. To maintain low latency at scale, most experts must remain loaded in GPUs even though only a small fraction are activated per token, driving memory and bandwidth demands far beyond the capacity of a few GPUs (e.g., at least 16 H100 GPUs to host DeepSeek-V3 [11]). Third, *attention and MoE layers have fundamentally different computational demands*. Compared with attention layers, MoE layers are more memory-intensive and their execution times are highly sensitive to expert activation patterns (see our analysis in §2.2).

These characteristics collectively induce a trilemma for MoE inference systems. Existing systems typically deploy the entire MoE model as a single monolithic instance and perform resource management and scaling at the model-instance level [2, 9, 30]. This coarse-grained design overlooks the heterogeneous resource demands of attention and MoE layers and often applies uniform resource configurations (e.g., identical parallelism degrees) across both layer types. Consequently, it requires to over-provision GPU resources to accommodate the peak demands of MoE layers—such as hosting all experts in GPU memory and handling extensive activations—even though attention layers require significantly fewer resources.

[1]* Equal contribution.
[2]† Corresponding author: Minchen Yu (yumincen@cuhk.edu.cn).

When combined with dynamic inference workloads, this approach results in substantial resource waste and low utilization. While recent works such as MegaScale-Infer [37] and xDeepServe [29] disaggregate attention and MoE layers onto separate clusters, they generally lack precise, fine-grained resource scaling and expert management, e.g., pinning a fixed set of experts to dedicated GPUs. This still results in significant resource inefficiency under dynamic inference workloads commonly observed in real-world scenarios [5, 14, 26, 32, 33].

An ideal MoE inference system should maximize resource efficiency and deliver the best performance per unit cost, e.g., per-GPU token generation throughput, while satisfying token-level SLO requirements. To this end, we propose JANUS , a resource-efficient and scalable MoE inference system. JANUS is built on two key insights. *First, MoE and attention layers should be disaggregated onto different sub-clusters due to their distinct resource demands.* This separation enables precise rightsizing of resource allocation and fine-grained expert management, which unlocks new opportunities for enhancing resource efficiency under dynamic workloads. *Second, MoE layers are generally memory-bound, and their execution time increases with the number of activated experts* (Fig. 2). Therefore, JANUS should balance the activated expert counts across GPUs for low execution latency and improved SLO attainment.

Following these insights, JANUS deploys attention and MoE layers on separate sets of GPU worker nodes, with each node hosting multiple attention or MoE serving instances. JANUS addresses three main challenges in disaggregated MoE inference. First, disaggregation introduces m-to-n communication between m attention instances and n MoE instances at every layer, which significantly increases end-to-end inference latency. As this overhead is dominated by many small data transfers, JANUS trades a modest increase in aggregate data volume for fewer, larger transfers. It employs an *adaptive two-phase transmission scheme* that first aggregates intermediate data from intra-node instances and then performs bulk transfers to destination nodes. This design significantly reduces the number of data transfers between two sub-clusters and, in turn, reduces overall inference latency.

Second, decoupling attention and MoE greatly expands the number of GPU workers available to host experts, but also introduces a complex layer-wise scheduling problem—determining, at inference time, how expert activation requests for each layer should be distributed among MoE instances to minimize inference latency. This activation request scheduling must incur extremely low overhead to meet the stringent latency requirements of layer-wise MoE computation, which typically completes within only a few hundred microseconds (see Fig. 1). To address this challenge, JANUS introduces an *activation load-balanced scheduler* that uses a fast heuristic algorithm to minimize the number of activated experts per MoE instance. JANUS implements the algorithm as a GPU kernel, eliminating CPU–GPU data synchronization, and performs scheduling in a fully distributed manner without cross-GPU coordination. Consequently, JANUS sustains the scheduling and synchronization overhead at the microsecond level, effectively meeting the stringent latency requirements of layer-wise MoE execution.

Table 1: Memory footprint of state-of-the-art MoE models.

Model	**Expert Mem. (GB)**	**Total Mem. (GB)**	**Ratio (%)**
Qwen3-235B [31]	423.0	438	96.5
DS-V2 [10]	421.0	472.0	89.2
DS-V3/R1 [6, 11]	1258.0	1342.0	93.7
Grok-1 [28]	586.0	628.0	91.7

The third challenge is to maximize per-GPU throughput, i.e., performance per unit cost, while meeting SLO requirements. This necessitates carefully determining resource allocation for both layer types and deciding how experts are placed across MoE instances. JANUS employs a *activation-aware expert management scheme* that dynamically adjusts expert redundancy (i.e., the number of expert replicas) and their placement to minimize the expected number of co-activated experts per MoE instance. This scheme also periodically configures the optimal numbers of attention and MoE instances to achieve the best resource efficiency subject to SLO constraints. In conjunction with the activation scheduler, this design effectively reduces inference latency and improves SLO attainment.

We implement JANUS on top of SGLang [18] and evaluate it using state-of-the-art MoE models. Experiments show that JANUS improves per-GPU throughput by up to 3.9× over baselines while meeting SLO requirements. We further demonstrate that, thanks to its disaggregated design and core optimizations, JANUS can dynamically adapt to changing workloads by incrementally adjusting attention and MoE configurations, reducing inference resource cost by 25% compared with monolithic approaches. These results highlight that JANUS achieves both high scalability and resource efficiency.

2 Background and Motivation

2.1 MoE Inference

Recent state-of-the-art LLMs increasingly adopt the MoE architecture to expand model capacity while maintaining low per-token compute cost [11, 29, 31]. A typical MoE model consists of tens of layers in which attention and MoE layers are interleaved. Attention layers maintain key-value (KV) caches for subsequent token generation, whereas MoE layers are stateless and comprise a gating network with multiple parallel experts. For each token, the gating network activates only a small subset of experts (e.g., top-k by gating score), thereby preserving bounded per-token computation.

Driven by these capacity advantages, modern MoE models have grown substantially in size, with expert parameters

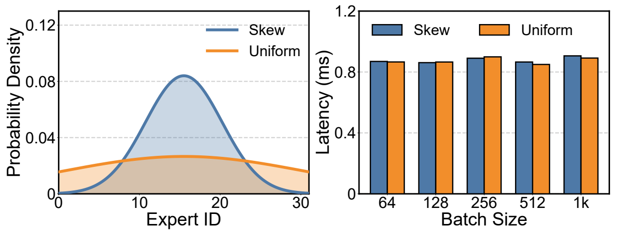

Figure 1: Two distributions of expert activation (left) and latency of an MoE layer under the activation patterns (right).

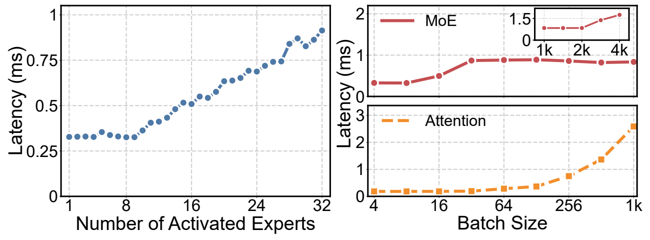

Figure 2: Performance of MoE and attention. Latency of an MoE layer under various number of activated experts (left) and the comparison between MoE and attention layers (right).

accounting for the majority (Table 1). For instance, DeepSeek-V3 employs 256 experts per MoE layer, and the experts collectively occupy 93.7% of the model's total memory footprint [11]. Serving such large MoE models entails substantial GPU resources—fully loading the model parameters of DeepSeek-V3 requires at least 16 H100 GPUs. Meanwhile, online inference workloads typically exhibit highly dynamic request patterns [14, 33, 35], necessitating elastic resource provisioning to meet stringent performance requirements that are often specified as token-level SLOs (e.g., TPOT). Consequently, MoE inference systems must be highly scalable—they should satisfy SLO requirements under fluctuating workloads while maintaining high resource utilization [2, 7, 9, 37].

2.2 Characteristics of MoE Inference

To understand the computational demands of MoE inference, we analyze the characteristic of MoE and attention layers and highlight their differences.

Understanding MoE layers. We start with the theoretical analysis for MoE layers using roofline model [2, 27, 34]. In an MoE layer, the computation for an expert is dominated by two General Matrix Multiplication (GEMM) operations. Let d_h and d_e denote the hidden dimension and expert intermediate dimension, respectively. For an expert with batch size b (i.e., b tokens routed to that expert), its arithmetic intensity can be approximated as $I_e \approx 2bd_hd_e/2d_ed_h = b$. To operate in the compute-bound regime, the arithmetic intensity must exceed π/β, where π is peak FLOPs and β is memory bandwidth of the target hardware, i.e., $I_e \approx b \geq \pi/\beta$.

Consider the entire MoE layer with n experts under top-k uniform expert routing. The expected per-expert batch size is $b = B \cdot k/n$, where B is the total batch size entering the layer. Therefore, the minimal layer-wise batch size required is $B \geq \pi n/\beta k$. For instance, the H100 (A100) provides 989 (312) TFLOPs/s and 3.35 (2.0) TB/s of memory bandwidth. Under this roofline, DeepSeek-V3 would require a batch size of about 18k (5k) tokens to operate in the compute-bound regime on H100 (A100) GPUs, which is hardly achievable for many real-world inference scenarios where batch sizes are often below 100 [17, 22].

Takeaway #1: *MoE layers are generally memory-bound across typical online inference workloads.*

We further characterize the performance of MoE layers using the representative MoE model DeepSeek-V2 [10]. Specifically, we deploy 32 experts from a single MoE layer on one H100 GPU, emulating the layer-wise expert density used in actual full model deployment. Fig. 1 reports the execution latency of this 32-expert MoE layer under varying numbers of concurrent activation requests (i.e., batch sizes). For each batch size, we activate all experts following a Gaussian-distributed pattern: we vary the distribution's variance to control activation skewness, while ensuring that every expert is selected at least once. The results show that increasing the batch size has only a marginal impact on latency, consistent with memory-bound behavior. Moreover, latency remains nearly the same across both uniform and skew activation distributions, indicating that performance is determined primarily by the total number of activated experts rather than how these experts are activated.

Fig. 2 (left) further shows the latency of an MoE layer under different numbers of activated experts, with the batch size fixed at 64. Note that the latency increases approximately linearly with the number of activated experts; only when the number of activated experts is very small (e.g., below 9) does a near-constant kernel launch overhead dominate. Together, these results demonstrate that MoE layers are memory-bound and the number of activated experts is the primary determinant of MoE execution time.

Takeaway #2: *In the memory-bound regime, the execution latency of MoE layer scales linearly with the number of activated experts. In contrast, batch sizes and activation patterns have negligible impact on MoE performance.*

Comparison between MoE and attention layers. Attention layers are a fundamental component of transformer-based LLMs. Prior work has extensively characterized their performance: the prefill phase is largely compute-bound, while decoding—generating each subsequent token—is predominantly limited by memory bandwidth [1, 36]. In this paper, we primarily focus on the decoding phase, which typically dominates end-to-end inference latency. A wide range of approaches have been proposed to disaggregate prefill and decode phases for tailored optimization [17, 35, 36]; these techniques are orthogonal to JANUS and can be readily integrated. We provide a more detailed discussion in §6.

Figure 2 compares the decoding latency of attention and MoE layers in DeepSeek-V2 across different batch sizes, with input length fixed at 512, following prior work [25]. In this experiment, a single H100 GPU hosts 32 experts, and each token activates exactly one expert in a balanced top-1 routing configuration. Consistent with our earlier observations, the MoE layer remains memory-bound until the batch size exceeds 2k and exhibits a two-stage latency pattern. For small batches (batch size < 32), latency is primarily determined by the number of activated experts, and beyond that it remains largely flat until the transition to the compute-bound regime at very large batches. In contrast, the latency attention layers remains stable under small-to-moderate loads, with a sharp rise when the batch size exceeds 128.

Takeaway #3: *MoE and attention layers exhibit distinct latency patterns; consequently, their optimal resource configurations and batching strategies differ significantly.*

2.3 Limitations of Existing Solutions

A wide range of systems have been proposed to serve large MoE models efficiently [2, 7, 9, 18, 24]. These solutions generally adopt a monolithic, coarse-grained model deployment and resource management. They deploy the entire MoE model across a set of GPUs as a single serving instance, where attention and MoE layers share identical resource configurations. To accommodate the large expert memory footprint, experts are typically sharded across multiple GPUs via expert parallelism (EP). Hence the degree of attention-side parallelism (e.g., data parallelism, or DP) is configured to match EP in many real-world deployments like DeepSeek-V3. This tightly coupled model-level design leads to two major limitations.

Limited scalability. In the monolithic design, a full MoE model instance is the basic unit of management, which fundamentally constrains scalability. Specifically, it requires replicating the entire model to create additional serving instances. This introduces substantial startup and warm-up overhead for large MoE models, as each replica requires loading all model parameters across a large number of GPUs. An alternative is to increase the degree of parallelism for an existing instance (e.g., adjusting DP and EP). This typically requires disruptive stop-and-restart procedures, again involving re-partitioning and reloading the full model on all participating GPUs. Both approaches incur high overhead and fail to elastically scale MoE inference in response to dynamic workloads.

Resource inefficiency. The monolithic approach also ignores the distinct performance characteristics of attention and MoE layers (§2.2) and couples their resource configurations, which substantially impairs overall utilization. Compared with attention layers, MoE layers are generally more resource-intensive, requiring a large number of GPUs to host all experts and sustain low execution latency. Under a shared DP and EP configuration, scaling resources to satisfy MoE demands (e.g., expert memory and activation throughput) automatically increases the allocation to attention layers, even when they do not need additional capacity (see §5.2). This coupled scaling leads to low resource utilization and unnecessarily high serving cost.

Recent works have explored disaggregating attention and MoE layers and deploying them on distinct sub-clusters [25, 29, 37]. However, these systems generally lack precise, fine-grained resource allocation and expert management, and thus the fundamental issue of resource inefficiency remains largely unsolved. For example, xDeepServe [29] simply pins each expert in a layer to a dedicated device for large-scale expert parallelism. Considering the dynamic workloads in many real-world LLM inference [23, 26, 33], such static strategies lead to resource inefficiency and poor utilization in these scenarios.

2.4 Key Insights and Challenges

To address the aforementioned limitations, we propose JANUS, a system for scalable and resource-efficient MoE inference. JANUS is built on two key insights.

First, attention and MoE layers should be disaggregated and deployed separately due to their distinct performance characteristics and resource requirements. Disaggregation enables fine-grained, module-specific scaling, e.g., independently adjusting resources and parallelisms for the two layer types. Compared with monolithic designs, this also reduces scaling overhead, as the system can incrementally resize the attention or MoE sub-clusters without restarting or reconfiguring an entire model instance.

Second, balancing the number of activated experts across GPUs is critical to improving MoE inference performance. Under typical online inference batch sizes (e.g., <100), attention latency remains relatively stable, whereas MoE latency is more dynamic and strongly correlated with expert activations (Fig. 2), offering greater optimization opportunities in end-to-end inference. Consequently, a key optimization strategy is to minimize and balance the number of activated experts per GPU, enabling JANUS to significantly reduce end-to-end MoE latency and improve SLO attainment.

However, realizing the two insights in JANUS introduces three main challenges. *(1) Communication overhead:* In disaggregated MoE inference, attention and MoE instances must exchange activations at every layer, resulting in frequent m-to-n communication with high overhead. *(2) Expert activation load balancing:* To minimize execution latency, JANUS must judiciously schedule expert activation requests across MoE instances to balance the number of activated experts per GPU. This scheduling must incur extremely low overhead to meet the microsecond-level latency requirement of MoE layer execution. *(3) High resource efficiency:* Key to MoE inference is to improve resource efficiency, maximizing per-GPU throughput while meeting SLO requirements. To this end, JANUS

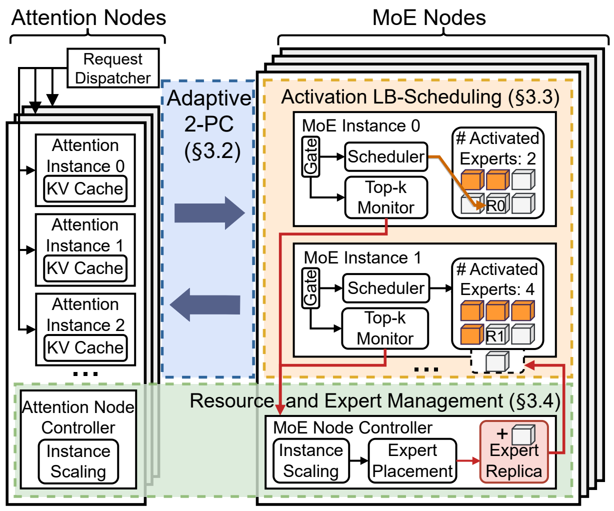

Figure 3: Architecture overview of JANUS.

must carefully determine how many resources to allocate to attention and MoE sub-clusters, and how to replicate and place experts across MoE instances.

3 System Design

In this section, we introduce the system design of JANUS and elaborate how it addresses the three main challenges.

3.1 Overview

Fig. 3 shows the architecture of JANUS. The cluster consists of two types of GPU nodes—attention nodes and MoE nodes—which are independently managed and scaled within two distinct sub-clusters. JANUS employs an adaptive two-phase communication mechanism for low-latency data transfers between attention and MoE nodes (§3.2). On the attention side, each attention node hosts multiple attention instances. Each instance maintains a full replica of all attention layers (i.e., DP) and runs on a single GPU [1]. Additionally, a request dispatcher routes incoming requests across attention instances, and an attention-node controller dynamically scales the number of instances based on the workload.

For a MoE node, it similarly runs multiple MoE instances, each on a single GPU and hosting multiple experts. Within each MoE instance, a lightweight activation scheduler decides whether and how incoming expert activation requests are served using local experts. All MoE instances collectively perform this scheduling in a fully distributed manner to balance the number of activated experts per instance and minimize inference latency, without incurring cross-GPU synchronization (§3.3). MoE instances also collect expert activation statistics and report them to a MoE-node controller, which determines instance scaling, expert redundancy, and expert placement. Together with the attention controller, this enables coordinated, fine-grained resource management across attention and MoE sub-clusters (§3.4).

We next present the three key designs of JANUS that address the challenges highlighted in §2.4.

3.2 Adaptive Two-Phase Communication

While there are a wide array of cluster-wide collective communication solutions, such as NCCL [16] and MSCCL++ [19], they are ill-suited for disaggregated MoE inference due to two reasons. First, communication between attention and MoE instances is inherently *asymmetric many-to-many*. With m attention instances and n MoE instances, each attention instance typically needs to send intermediate data of a layer to all MoE instances, and vice versa, yielding an m-to-n communication pattern. In contrast, standard collective primitives (e.g., all-reduce and all-gather) assume a symmetric group in which all participants contribute and consume data in a uniform manner. Second, these collective mechanisms are typically configured for a fixed communication group and are not designed to handle dynamic group sizes. In disaggregated MoE inference, the number of attention and MoE instances can change over time as the system elastically scales sub-clusters in response to workload fluctuations. Moreover, recent expert-parallel communication solutions such as DeepEP [4] are designed for monolithic MoE inference, inheriting these limitations and remaining inapplicable to disaggregated settings.

JANUS implements an efficient communication mechanism tailored for disaggregated MoE inference. We note that each individual data transfer between attention and MoE nodes is small, but the sheer frequency of these transfers dominates communication overhead. Accordingly, JANUS prioritizes reducing *the number of transfers* rather than the aggregate data volume, and introduces two key designs to minimize end-to-end communication latency.

Gating on the MoE side. JANUS places the gating network on the MoE side to reduce communication overhead. As shown in Fig. 4 (left), a straightforward approach is to place the gate on the attention side so that only tokens actually routed to a given expert are transmitted, thereby reducing the volume of activation data sent over the network. However, this design introduces several additional costs. First, it requires transmitting extra routing metadata (e.g., top-k expert IDs and weights) alongside activations. This either multiplies the number of small transfers or requires (de)serialization and packing to combine activations and routing metadata into larger payloads. Second, selectively sending only the routed activations forces attention nodes to re-organize activation tensors (e.g., via memory re-layout), introducing extra GPU and CPU overhead. Although attention-side gating reduces

[1] JANUS primarily targets state-of-the-art MoE models with a large number of small-to-moderate-sized experts, where DP and EP dominate [11]. JANUS can also support tensor parallelism; we discuss this in §6.

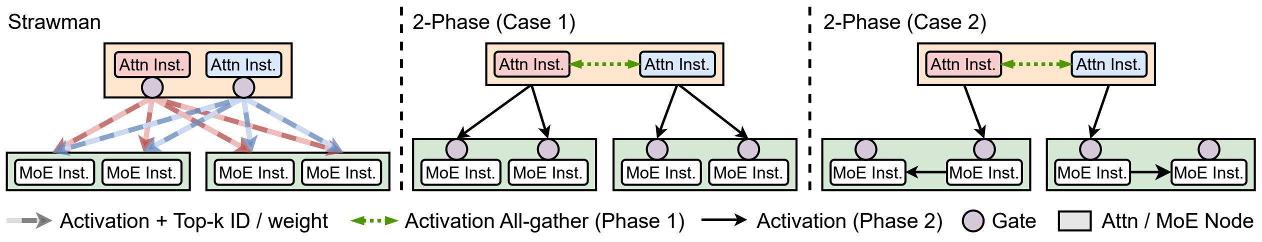

Figure 4: Comparison between a strawman solution (left) and adaptive two-phase communication (middle and right).

aggregate data volume, it increases complexity and leads to more fine-grained transfers, which is inefficient in our setting where the number of small data dominates end-to-end communication cost. Therefore, JANUS sends complete activations and performs gating on the MoE side, simplifying communication patterns and reducing the overall number of transfers.

Adaptive two-phase communication. Fig. 4 (left) illustrates a straightforward design where each attention instance directly communicates with every MoE instance. This approach scales poorly: the number of point-to-point transfers grows as $O(m \times n)$ with the numbers of attention (m) and MoE (n) instances, and the overhead is dominated by many small messages. To address this, JANUS introduces a two-phase communication scheme that exploits high-bandwidth intra-node NVLink to aggregate traffic and reduce the number of cross-node transfers (Fig. 4 middle and right). In the *first phase*, multiple attention instances on the same node perform an local aggregation of intermediate results (i.e., activations), using fast NVLink-based collective primitives (e.g., all-gather). After this phase, attention instances hold a full copy of the activations needed for the subsequent MoE computation. In the *second phase*, attention instances send activations to MoE instances. JANUS supports two regimes. **Case-1:** When each attention node needs to send data to only a small number of MoE nodes (e.g., 1–2 destinations), attention aggregators directly transmit the corresponding activation batches to the target MoE nodes. **Case-2:** When the number of MoE destinations or the data volume is large, JANUS switches to a one-to-one transmission pattern. Each attention instance sends its aggregated activations to a MoE instance in a designated node, which then distributes the data to other local instances via intra-node multicast over NVLink. JANUS adaptively selects between these two regimes based on resource configuration and traffic load. Communication in the reverse direction (MoE to attention) follows the same two-phase principle, using intra-node all-reduce to aggregate intermediate results on the MoE side before sending them back to attention nodes.

3.3 Activation Load-Balanced Scheduling

A key objective in JANUS is to minimize the number of activated experts per MoE instance, which requires judicious

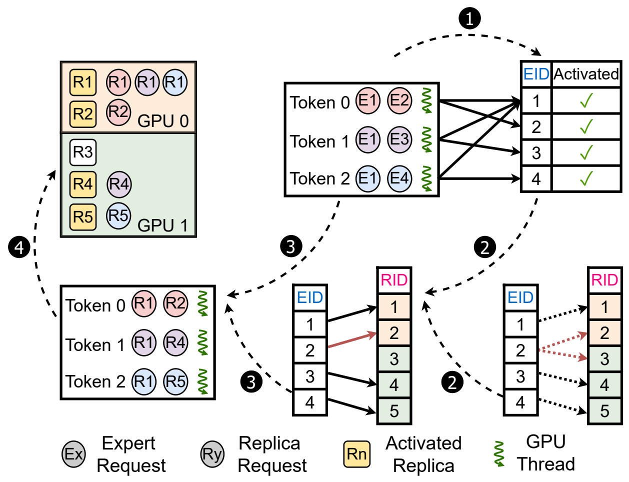

Figure 5: Scheduling workflow of JANUS.

expert-activation scheduling across MoE instances at every MoE layer. However, MoE layer execution typically completes within only a few hundred microseconds according to our measurements (Fig. 1), making such scheduling extremely challenging practice. First, *from an algorithmic perspective, finding the optimal schedule is a combinatorial load-balancing problem over all possible assignments of activation requests to MoE instances.* Computing the exact optimum is prohibitively expensive to perform online at every MoE layer. Second, *executing such scheduling incurs substantial synchronization overhead at the system level*. It requires collecting fine-grained information (e.g., per-token top-k activations), which involves frequent CPU–GPU synchronization, and coordinating across all participating MoE instances to perform global scheduling. This synchronization and coordination introduce non-trivial overhead that can easily dominate MoE execution time.

Scheduling workflow. JANUS introduces an efficient, microsecond-scale activation scheduling scheme. Fig. 5 illustrates the overall workflow. It first utilizes the token–expert routing results (top-k) to scans the batch and builds the set of distinct activated logical experts. This is implemented as a GPU kernel, and each token is processed in parallel by a GPU thread. Second, given the activated expert set and the expert–replica mapping, Janus runs the load-balancing algorithm to decide which replica should serve each logical expert. It assigns each activated expert request to the least-loaded instance among those hosting its replicas, balancing the number of activated experts per GPU. Third, it then remaps token requests from logical expert IDs to physical replica IDs, producing a per-token replica activation list. Finally, per-token activations are dispatched to the corresponding MoE instances (GPUs) for parallel execution. In this example, Janus chooses to serve requests for expert E2 using replica R2 instead of R3, because this choice better balances the number of activated expert replicas across GPUs, rather than simply balancing the number of activations.

Activation load-balanced algorithm. Janus employs an Activated-Expert-Balanced Scheduling (AEBS) algorithm that greedily, yet efficiently, reduces the number of activated experts per MoE instance (Algorithm 1). The key idea is to assign activation requests to expert replicas so that the maximum per-instance load—measured by the number of activated experts—is minimized. The algorithm first collects the set of all experts activated by the current batch of tokens (line 1). It then assigns single-replica experts to their unique hosting instances and schedules multi-replica experts to the least-loaded instances (lines 2-11). This yields near-balanced expert activation across instances while incurring only negligible computational overhead.

Synchronization-free scheduling. To avoid the overhead of global coordination, Janus makes AEBS fully synchronization-free across MoE instances via two mechanisms. First, Janus *implements AEBS as a GPU kernel to achieve microsecond-level scheduling delay*. It eliminates CPU–GPU synchronization for accessing per-token top-k activation data and enables parallel processing of large batches: GPU threads collaboratively perform expert collection, load tracking, and replica mapping for many tokens in parallel (i.e., step 1 and 3 in Fig. 5). Second, Janus *trades a small amount of redundant computation for eliminating cross-instance synchronization.* Rather than relying on a centralized global scheduler, each MoE instance independently runs the same AEBS kernel using identical inputs (token activation patterns, replica layout, and instance metadata). Because the algorithm is deterministic with respect to these inputs, all instances compute the same global assignment of activations to replicas. Janus propagates updated metadata (e.g., replica layout) only when the MoE sub-cluster is reconfigured—which occurs at a much coarser time scale than per-layer execution (e.g., on the order of hours)—and thus the associated overhead is negligible (§3.4). Moreover, the extra scheduling computation performed redundantly on each GPU is tiny relative to the cost of MoE forward passes. As a result, Janus eliminates inter-GPU synchronization and communication for activation scheduling while preserving correctness and imposing negligible overhead.

Algorithm 1 Activated-Expert-Balanced Scheduling

```
Input:
– $T$: number of tokens, $N$: number of instances
– $k$: number of activated experts per token
– $L(i,j)$: logical expert ID of the $j$-th activated expert for token $i$
– $R(e)$: number of replicas for expert $e$
– $\mathcal{G}(e)$: set of instances hosting replicas of expert $e$
– $P(e,g)$: physical replica ID of expert $e$ on instance $g$
Output:
– $O(i,j)$: physical replica ID of the $j$-th activated expert for token $i$
 1: $\mathcal{E} \leftarrow \bigcup_{i=1}^{T} \bigcup_{j=1}^{k} \{L(i,j)\}$        ▷ Collect all activated experts
 2: Initialize $activated\_replica[e] \leftarrow -1$ for all $e \in \mathcal{E}$
 3: Initialize $load[g] \leftarrow 0$ for all $g \in \{1,2,\dots,N\}$
    # Assign single-replica experts
 4: for all $e \in \mathcal{E}$ where $R(e) = 1$ do
 5:     $g \leftarrow$ unique instance in $\mathcal{G}(e)$    ▷ Only one hosting instance
 6:     $activated\_replica[e] \leftarrow P(e,g)$
 7:     $load[g] \leftarrow load[g] + 1$
    # Assign multi-replica experts via load balancing
 8: for all $e \in \mathcal{E}$ where $R(e) > 1$ do
 9:     $g^* \leftarrow \arg\min_{g \in \mathcal{G}(e)} load[g]$    ▷ Select least-loaded instance
10:     $activated\_replica[e] \leftarrow P(e,g^*)$
11:     $load[g^*] \leftarrow load[g^*] + 1$
    # Map tokens' activation requests to physical replicas
12: for $i = 1$ to $T$ do
13:     for $j = 1$ to $k$ do
14:         $O(i,j) \leftarrow activated\_replica[L(i,j)]$
```

3.4 Resource and Expert Management

In this section, we first describe Janus 's expert management mechanism, then present its fine-grained resource scaling. Together with the activation load-balanced scheduler from §3.3, these mechanisms reduce MoE-side latency, improve end-to-end SLO attainment, and enhance overall resource efficiency.

The core of expert management in Janus is *activation-aware replication and placement*. whose goal is to minimize the expected number of activated experts per instance and thus reduce inference latency. From our measurements, we observe that expert activations within a time window exhibit two patterns. *(1) Skewed popularity:* different experts are activated with markedly different frequencies. *(2) Co-activation:* certain pairs (or groups) of experts tend to be activated together for the same tokens. These patterns create two optimization opportunities: (1) allocate more replicas to hot experts to prevent them from becoming bottlenecks, and (2) place expert replicas so that frequently co-activated experts are spread across GPUs, thereby reducing the number of simultaneously active experts per GPU and lowering MoE layer latency.

Janus exploits this via an activation-aware expert replication and placement scheme that periodically reconfigures replica counts and placements based on recent activation statistics. This scheme operates at a coarse time scale (e.g., hours), and its decisions work in concert with the layer-wise activation scheduler to load-balance expert activations across MoE instances.

Expert replication. Given a certain MoE sub-cluster, i.e., a given number of MoE instances (GPUs) and expert slots per instance, JANUS first determines how many replicas to assign to each logical expert. Let there be E logical experts, N instances, and C expert slots per instance. The total number of physical replica slots is $S = N \times C$. Using activation counts collected over a sliding window, JANUS computes c_i, the number of activations of expert i. Each expert initially receives one replica, consuming E slots in total, and the remaining slots $S - E$ are used for redundancy.

Replica assignment is formulated as a slot-allocation problem, which aims to minimize per-replica expected load under capacity constraints of instances. Specifically, for expert i with r_i replicas, the per-replica load is $l_i = \frac{c_i}{r_i}$. JANUS identifies the expert with the largest l_i and allocates one additional replica to it in an iterative manner. This process repeats until all $S - E$ extra slots are exhausted. By continuously equalizing per-replica load in this way, hot experts receive more replicas, while cold experts remain single-replica. As a result, the expected activation pressure on any individual replica is reduced, lowering the probability that a single expert becomes a latency hotspot during inference.

Replica Placement. Given the replica assignments $\{r_i\}_{i=1}^{E}$ from the previous stage, JANUS then determines how these replicas should be placed across the N MoE instances, each with capacity C expert slots. The goal is to balance and minimize per-instance load, measured by the number of activated expert replicas.

Let $P(g)$ denote the set of (physical) expert replicas placed on instance g. Based on activation traces, JANUS estimates $a_{i,j}$, the co-activation frequency between logical experts i and j within a time window. Intuitively, when two experts with high $a_{i,j}$ are collocated on the same instance, the instance can be scheduled with many tokens activating both experts, increasing the number of concurrently active experts and thus MoE latency. We capture this effect via the *co-activation load* on instance g:

$$I(g) = \sum_{\substack{i,j \in P(g) \\ i<j}} a_{i,j} \tag{1}$$

The objective is then to place replicas so that the maximum co-activation load across GPUs is minimized. Formally, let $x_{i,g}$ be a binary variable and indicate whether a replica of logical expert i is placed on instance g. We aim to:

Algorithm 2 Activation-Aware Replica Placement

Input:
– N: number of instances, C: capacity per instance
– $\mathcal{R}$: set of replicas, l_i: load of replica i, e_i: logical expert of replica i
Output:
– $P(g)$: replicas assigned on instance g

```
 1: Initialize $P(g) \leftarrow \emptyset$, $slots[g] \leftarrow C$ for all $g \in \{1,2,\dots,N\}$
 2: Initialize $x_{e,g} \leftarrow 0$ for all experts $e$ and $g$
 3: Sort replicas $\mathcal{R}$ in decreasing order of $l_i$
 4: for all $i \in \mathcal{R}$ do
 5:     $G_i \leftarrow \{ g \in G \mid slots[g] > 0 \wedge x_{e_i,g} = 0 \}$
 6:     if $G_i \neq \emptyset$ then                      ▷ Slots feasible
 7:         $g^* \leftarrow \arg\min_{g \in G_i} \sum_{j \in P(g)} a(i,j)$
 8:         $P(g^*) \leftarrow P(g^*) \cup \{e_i\}$
 9:         $slots[g^*] \leftarrow slots[g^*] - 1$
10:         $x_{e_i,g^*} \leftarrow 1$
11:     else            ▷ No feasible slot; resolve via swapping
12:         $G_i^{\neg} \leftarrow \{ g \in G \mid x_{e_i,g} = 0 \}$ ▷ Instances without expert $e_i$
13:         $H_i \leftarrow \{ h \in G \mid slots[h] > 0 \}$ ▷ Instances with free slots
14:         Find $g \in G_i^{\neg}$, $h \in H_i$, and $j \in P(g)$ such that $x_{e_j,h} = 0$
    # Minimize co-activation load penalty of swapping
15:         $(g,j,h) \leftarrow \arg\min_{g,j,h} \Delta I(i \to g,\ j \to h)$
    # Apply replica swapping
16:         $P(g) \leftarrow (P(g) \setminus \{j\}) \cup \{i\}$
17:         $P(h) \leftarrow P(h) \cup \{j\}$
18:         $x_{e_j,g} \leftarrow 0$,   $x_{e_j,h} \leftarrow 1$,   $x_{e_i,g} \leftarrow 1$
```

$$\begin{aligned} \min_{\{x_{i,g}\}} \quad & \max_{g \in \{1,\dots,N\}} I(g) \\ \text{s.t.} \quad & \sum_{i=1}^{E} x_{i,g} \le C, \\ & \sum_{g=1}^{N} x_{i,g} = r_i, \\ & x_{i,g} \in \{0,1\} \end{aligned} \tag{2}$$

Solving (2) exactly is combinatorial and complex. It can be reduced to an unrelated-machines scheduling problem, which is NP-hard [8]. Therefore, JANUS adopts a heuristic solution as shown in Algorithm 2. The key idea is to process replicas in descending order of load and, for each replica, choose the placement that incurs the smallest increase in co-activation cost while respecting capacity constraints. Concretely, the algorithm first initializes per-instance placement sets, remaining slots, and a bitmap indicating whether an instance already hosts a replica of a given logical expert (line 1–3). It then iterates over replicas in sorted order: if there exists an instance with free capacity that does not yet host that expert, the replica is placed on the instance that adds the least co-activation penalty (line 5–10). Otherwise, the algorithm performs a bounded swap between two instances to create a feasible placement with minimal increase in co-activation cost (line 11–18). This heuristic effectively spreads frequently co-activated experts across instances, approximating the min–max objective while remaining lightweight enough for periodic online reconfiguration.

Resource scaling. JANUS scales the numbers of attention and MoE instances to maximize performance per unit cost—measured by per-GPU throughput—while preserving SLO attainment. Each sub-cluster (attention and MoE) runs a lightweight controller that periodically collects request arrival and queueing statistics, per-token per-batch latency, and per-instance utilization. Based on these metrics, the controllers independently adjust their instance counts. When SLO violations begin to rise or queueing delays grow, JANUS uses a performance model to evaluate candidate scaling actions (e.g., adding attention or MoE instances). The model estimates whether a given action can recover SLOs and what additional GPU costs it incurs. JANUS then applies the least-cost action that is predicted to meet SLO targets. Symmetrically, when SLOs are consistently satisfied and utilization is low, the same model is used to identify safe scale-down actions that reduce GPU usage without violating SLOs. Because attention and MoE sub-clusters are disaggregated and managed independently, this scaling process no longer requires DP and EP to change in lockstep. As a result, JANUS can perform finer-grained, more cost-efficient capacity adjustments than monolithic designs.

4 Implementation

We implement JANUS on top of SGLang [18] with about 3.5K lines of Python and 300 lines of CUDA/C++ code, extending SGLang to support disaggregated MoE inference. For the attention sub-cluster, we reuse SGLang's mechanisms including request batching, dispatching, and KV-cache management. For cross-sub-cluster communication, we build the adaptive two-phase mechanism on UCX [20] and leverage GPUDirect RDMA for low-latency, zero-copy transfers between GPUs across nodes. During communication, receivers cache incoming data from multiple senders into a GPU staging buffer and incrementally accumulate it using reduce operations, allowing communication and computation to overlap. We empirically tune several UCX parameters (e.g., buffer sizes) for our workload characteristics. Intra-node collectives and reductions (e.g., all-gather and all-reduce over NVLink) are implemented using NCCL. For the MoE sub-cluster, each MoE instance runs the activation load-balanced scheduler as a GPU kernel, which performs parallel processing for each token in the batch.

5 Evaluation

In this section, we evaluate JANUS and answer the following questions: (1) What are the benefits of disaggregating attention and MoE layers in JANUS? (§5.2) (2) Can JANUS improve per-GPU throughput while meeting SLOs compared with state-of-the-art systems? (§5.3) (3) How does each individual design in JANUS contribute to its overall performance gains? (§5.4)

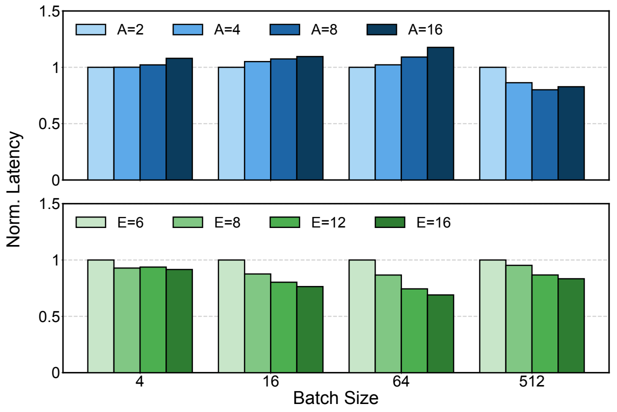

Figure 6: Latency of an attention (top) and MoE layer (bottom) under varying batch sizes and degree of parallelism.

5.1 Experiment Setup

Testbed. We deploy JANUS on a GPU cluster of up to 4 nodes. Each node is equipped with 8 NVIDIA H100, 80GB GPUs, 128 CPU cores, and 2 TB of host memory. Each GPU is connected with a 400 Gbps InfiniBand NIC. GPUs within a node are interconnected via 900 GB/s NVLink.

Models and workloads. We evaluate JANUS on DeepSeek-V2 [10], a representative large MoE model (Table 1), and on Scaled-DS, a set of scaled DeepSeek-style variants that modify the top-k activation, the number of experts, and per-expert sizes. We consider two Scaled-DS configurations. Scaled-DS-1 uses top-$k = 8$ among 160 experts per layer with a size of 1024 per expert, whereas Scaled-DS-2 also adopts top-$k = 8$ but enlarges the expert pool to 200 experts with a size of 1536 per expert. All model parameters and KV caches are stored in BF16 format. We use two representative workloads. First, we replay requests derived from the ShareGPT dataset [21], with average input length 16 tokens and output length 256 tokens. Second, we use BurstGPT [26] to synthesize realistic, dynamic inference arrivals that mimic production LLM services. Since the released contents are anonymized, we reconstruct request bodies with synthetic text while preserving the original arrival patterns.

Baselines. We compare JANUS against two baselines: *(1) SGLang*. SGLang is a state-of-the-art LLM inference system that represents the monolithic design. The entire MoE model is deployed as a single instance with a fixed parallelism configuration. The degrees of parallelism across both attention and MoE sides must match. SGLang scales only at coarse granularity: the model is deployed on 16, 32, 64 GPUs, etc., as a whole. *(2) DisAgg*. This baseline represents a disaggregated design. We implement a disaggregated MoE inference

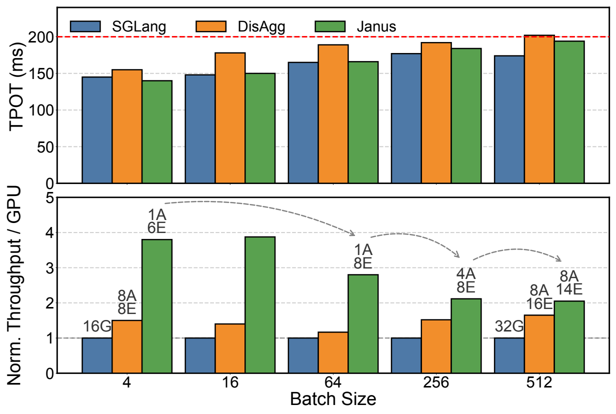

Figure 7: TPOT and per-GPU throughput of DeepSeek-V2 under different batch sizes. The TPOT SLO is fixed at 200 ms, requiring all systems to scale as load increases. Annotations (e.g., 1A6E) indicate the optimal configuration of attention (A) and expert (E) instances selected by JANUS.

system on top of JANUS 's codebase, but replace JANUS 's activation scheduler with random expert scheduling, a common strategy in existing disaggregated frameworks such as xDeepServe [29]. Additionally, DisAgg uses coarse-grained resource management, scaling resources in units of nodes (8 GPUs each) rather than at the instance level. We primarily evaluate resource efficiency and latency, measured as throughput per GPU and TPOT, respectively.

5.2 Benefits of Disaggregated Design

We first study the benefits of JANUS 's disaggregated design. Fig. 6 shows normalized latency for attention (top) and MoE (bottom) layers of DeepSeek-V2 under different batch sizes and instance counts. For attention, the effect of parallelism is strongly workload-dependent. Under low-to-moderate loads (batch size 4–64), increasing the number of attention instances actually increases latency, as synchronization and communication dominate the modest compute. In this regime, adding more instances both hurts latency and severely under-utilizes GPUs. Only when the batch size reaches 512 (compute-intensive) does a larger instance number reduce latency—at the expense of substantially higher GPU consumption. By contrast, MoE layers benefit more consistently from increasing the number of MoE instances, since their latency is dominated by the number of activated experts and memory bandwidth. More MoE instances provide more aggregate memory and bandwidth, reducing latency, but again at a higher cost.

Overall, there are two takeaways. First, attention and MoE layers exhibit fundamentally different scaling behaviors, motivating JANUS 's disaggregated design, which independently right-sizes attention and MoE sub-clusters to track real-time load. Second, no single static configuration is latency- and cost-optimal across workloads for both layer types, highlighting the need for fine-grained scaling to balance performance and resource efficiency.

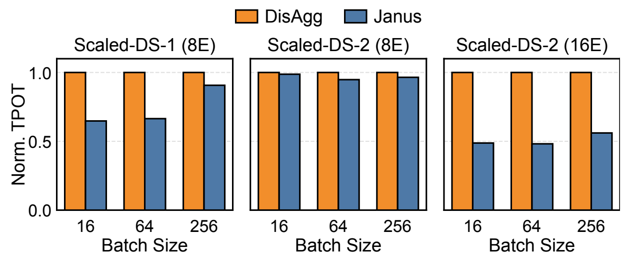

Figure 8: Normalized TPOT under various model variants and batch sizes.

5.3 End-to-End Performance

High Per-GPU throughput with SLO attainment. We evaluate the end-to-end performance of JANUS against SGLang and *DisAgg*. Fig. 7 reports TPOT latency and normalized per-GPU throughput across batch sizes. Fig. 7 (top) shows that JANUS consistently satisfies the 200ms TPOT SLO at all batch sizes and achieves latency comparable to SGLang—the non-disaggregated baseline—thanks to our optimized communication design. In contrast, *DisAgg* suffers from substantial communication overhead and imbalanced expert activation, failing to meet the SLO at the largest batch size (512). Fig. 7 (bottom) highlights the performance gains enabled by JANUS 's module-specific, fine-grained resource scaling. By dynamically selecting the ratio of attention to expert instances, JANUS improves per-GPU throughput by up to 3.9× and 2.8× over SGLang and *DisAgg*, respectively.

We further examine how JANUS and the baselines scale across batch sizes. Under light load, the baselines perform coarse-grained scaling and substantially over-provision attention capacity. In contrast, JANUS selects a minimal attention configuration (e.g., `1A6E` at batch sizes 4 and 16), concentrating GPUs on experts and achieving higher per-GPU throughput than SGLang. As load increases, JANUS progressively scales up attention capacity (e.g., `1A8E` at batch size 64, `4A8E` at 256, and `8A14E` at 512) to avoid attention bottlenecks. In contrast, the baselines perform only a single coarse-grained scaling step, increasing the configuration from 16 GPUs (`8A8E`) to 32 GPUs (`8A16E`) at the largest batch size, which limits their ability to match JANUS 's throughput and resource efficiency. To summarize, JANUS 's fine-grained, independent scaling of attention and expert instances effectively contributes to higher per-GPU throughput and improved serving efficiency.

Performance of model variants. We further evaluate JANUS against *DisAgg* on the Scaled-DS variants. In Fig. 8 (left), Scaled-DS-1 achieves considerable performance gains at low to moderate batch sizes because its smaller per-expert foot-

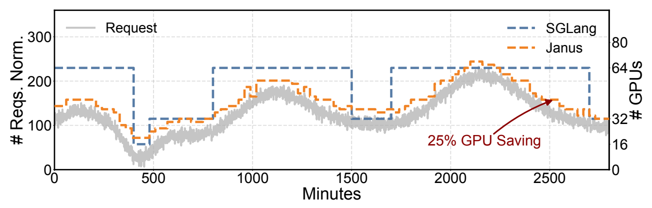

Figure 9: Simulation of scaling decisions for JANUS and SGLang under real-world workloads.

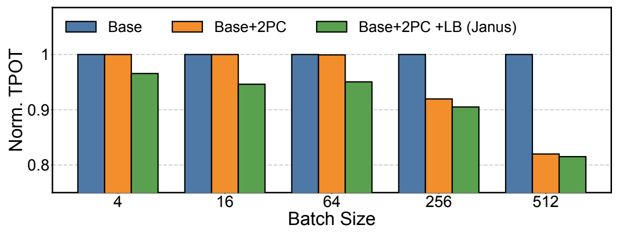

Figure 10: Performance breakdown of JANUS's key designs.

print allows extra replicas and thus more scheduling flexibility. However, this benefit diminishes as batch size grows and most experts are activated. For Scaled-DS-2 (Fig. 8, middle and right), the larger expert pool already saturates 8 MoE instances (8E), leaving no redundant replicas for JANUS 's scheduler and yielding only modest gains compared to Scaled-DS-1. Scaling out to 16 instances (16E) restores expert redundancy and enables substantial additional improvements from JANUS 's activation load-balanced scheduling.

Real-world workloads. Fig.9 compares the behavior of JANUS and SGLang under long-running, dynamic workloads using a two-day trace derived from real-world traffic [26]. Since we cannot access a large GPU cluster for such an extended period, we simulate each system's scaling behavior and set the scaling decision interval to 30 minutes. SGLang can only use three fixed resource configurations (16, 32, and 64 GPUs), whereas JANUS scales the numbers of attention and MoE instances at much finer granularity. As a result, JANUS more closely follows the diurnal fluctuations in request volume and continuously adapts its GPU allocation, avoiding long periods of over-provisioning. This elastic behavior reduces overall GPU consumption by 25% relative to SGLang while maintaining service quality.

5.4 Microbenchmarks

Performance breakdown. To quantify the contribution of each component in JANUS, we conduct an ablation study on TPOT latency across three configurations. (1) *Base*: a disaggregated implementation for attention and MoE layers without any JANUS optimizations. (2) *Base+2PC*: *Base* augmented with adaptive two-phase communication. (3) *Base+2PC+LB*: *Base+2PC* further enhanced with load-balanced scheduling

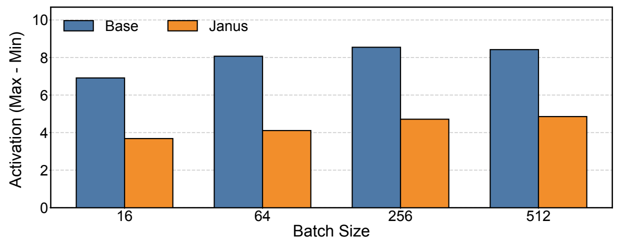

Figure 11: Difference in the number of activated experts between the most- and least-activated instances.

and activation-aware expert placement, corresponding to the full JANUS.

Fig. 10 presents results across varying batch sizes. First, the two-phase communication (+2PC) substantially alleviates the communication bottleneck inherent in disaggregated architectures under heavy workloads. At a batch size of 512, *Base+2PC* reduces TPOT latency by 18% relative to *Base*, indicating that optimizing the cross-sub-cluster data transfers is critical for scalability at high data volumes. Second, the load-balanced scheduling and expert placement (+LB) further optimizes performance by mitigating expert load imbalance, which becomes the dominant source of latency variability once communication is no longer the primary bottleneck. When the batch size exceeds 256, the performance gap between *Base+2PC* and *Base+2PC+LB* narrows because the larger token volume activates nearly all experts across nodes, leaving limited headroom for additional balancing. In contrast, the benefit of scheduling is more pronounced in the low-to-medium batch-size regime (e.g., batch sizes 16 and 64), where *Base+2PC+LB* achieves an additional 7% latency reduction over *Base+2PC*.

Effects of JANUS's scheduling. Fig. 11 reports the activation imbalance across MoE instances, measured as the difference in the number of activated experts between the most- and least-activated instances at each batch size. A smaller difference indicates a more balanced load and thus better performance. Across all batch sizes, *Base* exhibits substantially larger gaps (about 8 activations) than JANUS, indicating that some instances are heavily overloaded while others are underutilized. JANUS 's scheduling consistently reduces this gap to about 4 activations, effectively cutting the imbalance. This demonstrates that JANUS 's load-balanced scheduling can distribute the number of activated experts more evenly across instances, which in turn reduces stragglers and contributes to the latency improvements observed in Fig. 10.

Overhead of JANUS's scheduling. Fig. 12 reports the scheduling overhead of JANUS across different batch sizes and MoE sub-cluster scales (8 and 16 GPUs). In all settings, the overhead remains consistently below 100μs and grows only marginally as the batch size increases from 16 to 512. Similarly, doubling the number of GPUs from 8 to 16 introduces only a small, nearly constant offset without any notice-

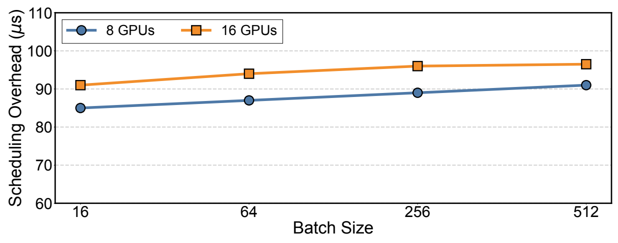

Figure 12: Overhead of JANUS's scheduling.

able growth trend. These results demonstrate that JANUS 's scheduling mechanism incurs negligible overhead and does not become a bottleneck even at large batch sizes or larger deployments.

6 Discussion and Related Work

Support for heterogeneous hardware. Modern data centers increasingly comprise heterogeneous accelerators, mixing different GPU generations or types [13, 15]. JANUS naturally accommodates such environments by allowing attention and MoE instances to be mapped to different hardware pools. Each instance type can thus be assigned to devices that best match its compute and memory characteristics (e.g., placing MoE on memory-rich GPUs), enabling additional cost savings and higher utilization, in line with prior work on disaggregated MoE inference over heterogeneous hardware [25,37]. In these settings, JANUS 's core mechanisms—two-phase communication, activation load-balanced scheduling, and activation-aware expert management—remain directly applicable.

Support for other parallelism schemes. JANUS can be naturally extended to work with additional intra-model parallelism schemes beyond DP and EP, such as tensor parallelism (TP). In our design, an instance is an abstract resource unit; each instance can internally be implemented as a tensor-parallel group of GPUs. The control plane only reasons about the aggregate capacity and cost of each instance, and can directly apply JANUS 's scheduling and resource management mechanisms on top of this abstraction. As a result, one can compose DP, EP, and TP in whatever configuration their model requires, while still benefiting from JANUS 's fine-grained, module-independent scaling.

Pipelining across attention and MoE. JANUS can further improve performance by enabling pipelined execution via micro-batching [29, 37]. Rather than processing an entire batch sequentially, the incoming request stream is partitioned into smaller micro-batches that flow through the pipeline. Attention and MoE modules then operate on different micro-batches concurrently, overlapping computation across modules and increasing hardware utilization, especially at large batch sizes. Since JANUS already tracks per-module loads and latency metrics, micro-batching can be simply incorporated while sustaining a low overhead.

Disaggregated LLM Inference. Disaggregation is an emerging design trend for LLM inference, which enables module-specific resource management and independent scaling across different stages of the serving pipeline. A line of work focuses on prefill-decoding disaggregation, where prefill and decoding requests are served by separate GPU pools with tailored configurations [17,36]. This direction is largely orthogonal to JANUS: in principle, JANUS can be applied within each pool to independently configure and scale attention and MoE resources for prefill and decoding, further improving utilization and cost efficiency.

Recent work has explored disaggregating attention from MoE or FFN layers to better match their distinct resource demands [3, 12, 29, 37]. For example, MegaScale-Infer [37] and Step-3 [25] decouple these modules onto heterogeneous hardware and introduce CPU-coordinated communication libraries that minimize GPU SM contention. EaaS [12] provides scalable, GPU-centric communication mechanisms for attention-expert disaggregation, while xDeepServe [29] demonstrates large-scale MoE inference on NPU superpods by placing each expert on a separate NPU and relying on a specialized data plane for efficient cross-device transfers. JANUS is orthogonal to these systems and differs in two key aspects. First, JANUS focuses on microsecond-level, fine-grained activation scheduling and expert placement, targeting dynamic or moderate-load regimes where static or coarse-grained policies leave significant headroom. Our activation load-balanced scheduling and expert management mechanisms can be integrated into existing systems to improve resource utilization. Second, JANUS introduces an adaptive two-phase communication scheme that tailors the m-to-n data-exchange pattern, complementing prior work on data-plane optimizations. JANUS can readily exploit these optimized communication libraries as its data plane to further enhance overall performance.

7 Conclusion

We presented JANUS, a scalable MoE inference system that disaggregates attention and MoE layers into distinct GPU sub-clusters. JANUS targets maximizing resource efficiency and per-GPU throughput while meeting token-level SLOs. To this end, it optimizes cross-sub-cluster communication and minimizes the number of activated experts per GPU to reduce inference latency, and it supports fine-grained, module-specific scaling to adapt capacity to workload dynamics. Evaluation shows that JANUS improves per-GPU throughput by up to 3.9× over state-of-the-art systems with SLO attainment.

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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

[2] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. Moe-lightning: High-throughput moe inference on memory-constrained gpus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, pages 715–730, New York, NY, USA, 2025. Association for Computing Machinery.

[3] Shaoyuan Chen, Wencong Xiao, Yutong Lin, Mingxing Zhang, Yingdi Shan, Jinlei Jiang, Kang Chen, and Yongwei Wu. Efficient heterogeneous large language model decoding with model-attention disaggregation. *arXiv preprint arXiv:2405.01814*, 2025.

[4] DeepSeek-AI. DeepEP. https://github.com/deepseek-ai/DeepEP, 2025.

[5] 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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, Santa Clara, CA, July 2024. USENIX Association.

[6] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

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

[8] Jan Karel Lenstra, David B. Shmoys, and Eva Tardos. Approximation algorithms for scheduling unrelated parallel machines. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 217–224, 1987.

[9] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed MoE training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 945–959, Boston, MA, July 2023. USENIX Association.

[10] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

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

[12] 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.

[13] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '25, pages 586–602, 2025.

[14] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, pages 1112–1127, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Zizhao Mo, Jianxiong Liao, Huanle Xu, Zhi Zhou, and Chengzhong Xu. Hetis: Serving llms in heterogeneous gpu clusters with fine-grained and dynamic parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*, pages 1710–1724, New York, NY, USA, 2025. Association for Computing Machinery.

[16] NVIDIA. Nvidia collective communications library (nccl). https://github.com/NVIDIA/nccl, 2025.

[17] 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 51st Annual International Symposium on Computer Architecture*, ISCA '24, pages 118–132. IEEE Press, 2025.

[18] SGLang. https://github.com/sgl-project/sglang, 2025.

[19] Aashaka Shah, Abhinav Jangda, Binyang Li, Caio Rocha, Changho Hwang, Jithin Jose, Madan Musuvathi, Olli Saarikivi, Peng Cheng, Qinghua Zhou, et al.

Msccl++: Rethinking gpu communication abstractions for cutting-edge ai applications. *arXiv preprint arXiv:2504.09014*, 2025.

[20] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. Ucx: an open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[21] ShareGPT Teams. https://sharegpt.com/, 2023.

[22] Jovan Stojkovic, Chaojie Zhang, Inigo Goiri, Josep Torrellas, and Esha Choukse. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1348–1362, Los Alamitos, CA, USA, March 2025. IEEE Computer Society.

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

[24] vLLM. https://github.com/vllm-project/vllm, 2025.

[25] 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.

[26] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '25)*, New York, NY, USA, 2025. Association for Computing Machinery.

[27] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[28] xAI. https://x.ai/blog/grok-os, 2024.

[29] Ao Xiao, Bangzheng He, Baoquan Zhang, Baoxing Huai, Bingji Wang, Bo Wang, Bo Xu, Boyi Hou, et al. xDeepServe: Model-as-a-service on Huawei CloudMatrix384, 2025.

[30] Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. Moe-infinity: Efficient moe inference on personal machines with sparsity-aware expert cache, 2024.

[31] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[32] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, Haoran Yang, et al. Torpor: Gpu-enabled serverless computing for low-latency, resource-efficient inference. In *Proceedings of the USENIX Annual Technical Conference*, 2025.

[33] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Yue Cheng, Wei Wang, Ao Wang, and Ruichuan Chen. λScale: Enabling fast scaling for serverless large language model inference. *arXiv preprint arXiv:2502.09922*, 2025.

[34] Sungmin Yun, Seonyong Park, Hwayong Nam, Younjoo Lee, Gunjun Lee, Kwanhee Kyung, Sangpyo Kim, Nam Sung Kim, et al. The new llm bottleneck: A systems perspective on latent attention and mixture-of-experts. *arXiv preprint arXiv:2507.15465*, 2025.

[35] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Blitzscale: fast and live large model autoscaling with o(1) host caching. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association.

[36] 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 Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association.

[37] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, et al. Megascale-infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, pages 592–608, New York, NY, USA, 2025. Association for Computing Machinery.