StreamEP: Straggler-Tolerant MoE Decoding without Communication Barriers

Yizhuo Liang*
University of Southern California

Shaoyu Wang*
University of Southern California

Jaeyong Song
Seoul National University

Yanqi Zhou
Google DeepMind

Geon-Woo Kim
The University of Texas at Austin

Guangrong He
University of Southern California

Seo Jin Park
University of Southern California

Abstract

Mixture-of-Experts (MoE) models use expert parallelism (EP) to distribute experts across GPUs, relying on collective communication that synchronizes every device at every layer. To minimize communication latency, hyperscalers provision abundant network bandwidth in data centers and deploy MoE-optimized stacks that require Hopper-generation GPU features. But on commodity GPU clusters with shared and nonuniform interconnects, this communication barrier serializes execution behind the slowest transfer.

We present *stream expert parallelism (StreamEP)*, a paradigm that removes the collective communication barrier entirely: each GPU independently receives tokens, executes the appropriate layer, and forwards results as they become ready. We build StreamInfer, a serving system that realizes StreamEP on vanilla NCCL. StreamInfer introduces per-layer token queues for asynchronous token management, a defragmenting scheduler that consolidates partial batches while guaranteeing forward progress, and a two-phase communicator for asynchronous and variable-size GPU-to-GPU transfers. On 16×L40S and 16×A100 clusters with 200 Gbps internode networking, StreamInfer achieves up to 1.7× higher decoding throughput than traditional EP, demonstrating that efficient large-scale MoE serving is achievable on commodity hardware.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; **Machine learning**.

Keywords: Mixture-of-Experts (MoE), Large Language Models (LLMs), Distributed Inference, Decoding, Expert Parallelism, Model Serving

*Equal contribution

SOSP '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/26/09
https://doi.org/10.1145/3830418.3843869

ACM Reference Format:
Yizhuo Liang, Shaoyu Wang, Jaeyong Song, Yanqi Zhou, Geon-Woo Kim, Guangrong He, and Seo Jin Park. 2026. StreamEP: Straggler-Tolerant MoE Decoding without Communication Barriers. In *Symposium on Operating Systems Principles (SOSP '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3830418.3843869

1 Introduction

Mixture-of-Experts (MoE) has become the mainstream architecture for frontier language models, and it has demonstrated strong results on code generation, reasoning, and instruction-following benchmarks [4, 16]. By activating only a small fraction of experts per token per layer, MoE models [4, 7, 8, 16, 21, 27, 30, 33] pack hundreds of billions to trillions of parameters while remaining practical in cost.

These models far exceed the memory of a single GPU, so they are sharded across many GPUs and servers for serving. Expert parallelism (EP) has become the standard approach: each GPU holds a subset of experts, and a pair of collectives, *dispatch* and *combine*, shuffles tokens to the right GPUs for expert computation and back at every MoE layer [21]. Implementations realize this pair as all-to-all exchanges [4, 48] or as all-gather and all-reduce collectives [49]. EP keeps kernel efficiency high during decoding, but either realization is a hard synchronization point: every GPU must finish its exchange before any can move on, so the *slowest link* (Figure 1(a)) or *most loaded device* drags the pace for each MoE layer. We call this *barrier EP*.

This straggler problem can be effectively hidden by over-provisioning the bandwidth of GPU interconnects. Hyperscaler data centers employ large NVLink domains (e.g. NVL72) where each GPU has up to 900 GB/s bandwidth for both send and receive. When the model deployment exceeds the NVLink domain, hyperscalers provision 400–800 Gbps of network bandwidth *per GPU*. In addition, they employ MoE-optimized software stacks [1, 23, 24, 47, 48] which overlap communication and computation through micro-batching, by exploiting Hopper or later GPU features. Together, the dedicated NICs, over-provisioned bandwidth, and the special software can minimize and hide communication stragglers.

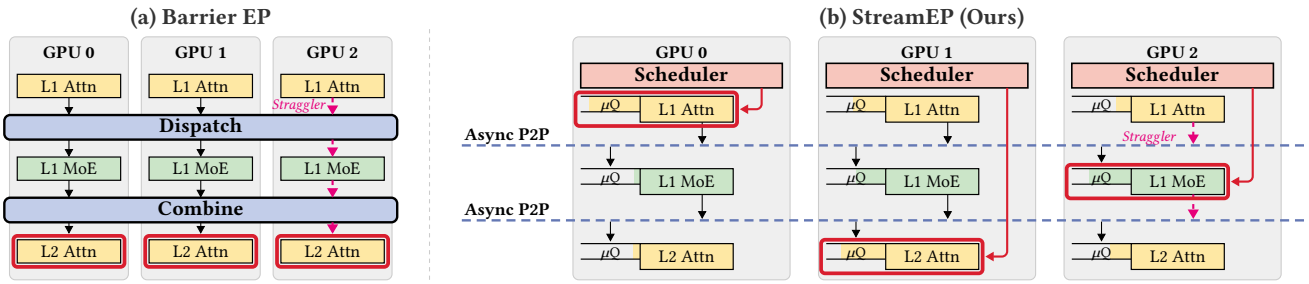

Figure 1. Barrier EP (a) vs. StreamEP (b), with identical expert placement and a straggling link on GPU 2 marked in pink. Red frames mark the module each GPU is currently executing. **(a)** Blocking Dispatch/Combine collectives run in lockstep, so every GPU stalls behind the straggler. **(b)** Outputs stream over async P2P into per-layer μ-queues, whose fill shows the tokens waiting in each. The scheduler on each GPU continuously picks a queue to drain and execute without waiting for the full batch.

Unfortunately, such fabrics and GPUs are out of reach for most users who need to serve MoE models on their own infrastructure for various reasons: data compliance, academic HPC pipelines, or customized open-source models with continual training or RL [41]. We call these deployments *commodity clusters*: multi-node GPU systems with communication fabrics constrained relative to hyperscaler infrastructure. They lack some combination of ultra-high-bandwidth intra-node GPU connectivity, high NIC bandwidth per GPU, and sufficient inter-node bisection bandwidth. These users often have pre-Hopper GPUs (e.g., A100, L40S) connected by 100–200 Gbps commodity networking. For instance, only 6% of GPUs available from NSF ACCESS providers meet the requirements for DeepEP [48], and the proprietary network fabrics of major cloud providers such as AWS and GCP support these optimized stacks only on specific instance and backend combinations (§2.2).

On such clusters, the collective barrier becomes the dominant cost, consuming up to 75% of each MoE layer's execution time [22]. Beyond the lower bandwidth itself, three factors cause communication stragglers and amplify the communication bottleneck:

- **Nonuniform and shared interconnects.** Intra-node links can be a mix of PCIe and NVLink, while each inter-node link is often shared by more than one GPU, so messages arrive at widely varying times even when the data to send/receive on each rank is perfectly balanced, as shown in Figure 2.
- **Communication load imbalance.** While expert placement can be decided to minimize MoE compute load imbalance on each rank via algorithms like EPLB [5], the imbalance of tokens to transfer [20] for each individual pair of EP ranks can still be significant (§B).
- **Transient network congestion.** Under co-tenancy, even a brief bandwidth dip on a single link stalls every GPU behind the barrier (§6.5).

Together, these create communication *stragglers* that the collective barrier forces every GPU to wait on.

This paper proposes *Stream Expert Parallelism (StreamEP)*, which unwraps the dispatch and combine collectives into individual point-to-point messages to avoid getting blocked by a straggler network link or GPU (Figure 1(b)). Instead of waiting on the straggler, each GPU now begins computation as soon as a partial batch arrives, without waiting for the full batch, akin to stream processing. Without collectives always returning the full batch, StreamEP accumulates the incoming partial batch tokens in *micro-queues* (μ-queues, also μQs), which buffer input for computation. A scheduler on each GPU independently decides which μQ to drain and execute in the next step, based on the queue depths. After execution, the output is forwarded to other GPUs via asynchronous p2p messages as well. The result is a natural pipeline: a GPU computes on tokens that have already arrived, sends results onward, and picks up the next arriving batch—tolerating both bandwidth heterogeneity and dynamic load skew without any explicit load balancing. Adaptive batching has been widely used to provide both low latency and high throughput. StreamEP achieves this naturally: incoming tokens queue while the GPU executes previously scheduled computation, yielding batches with sufficient arithmetic intensity without letting GPUs idle.

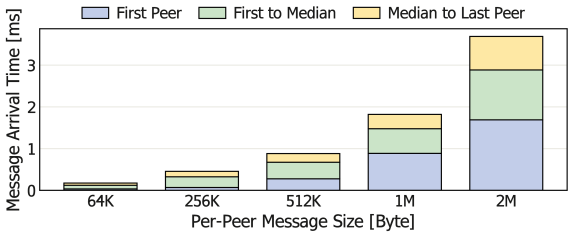

Figure 2. First peer, median, and last peer message arrival times inside an instrumented NCCL all-to-all collective on an 8-node, 16-GPU cluster with 200 Gbps RoCE on each node. More data is shown in §A. Sizes from 256 KB to 1 MB are about the per-peer message sizes GPT-OSS-120B generates in this cluster.

Unwrapping these collectives opens up new system challenges that we address with dedicated mechanisms:

- **Layer-level asynchronous execution.** Without collective synchronization, each GPU must independently manage tokens arriving asynchronously from multiple sources,

track each token's target layer, and schedule layer executions in an order that is no longer globally determined.

- **Batch reformation overhead.** At every layer boundary, token batches are split and reformed as tokens are routed to different GPUs for expert processing and back. This per-layer batch reconstruction incurs significant overhead from host-device metadata transfers and GPU memory management, particularly with paged attention's KV cache management [19].
- **Batch fragmentation.** Removing synchronization barriers causes token batches to fragment: tokens from a single logical batch arrive at different times across GPUs, resulting in many small, inefficient executions of the same layer. Naïve scheduling strategies either exacerbate this fragmentation or cause livelock with newly arriving requests.
- **Dynamic point-to-point communication.** Asynchronous execution makes communication patterns unpredictable: any GPU may send a variable-size token batch to any other GPU at any time. Yet GPU-direct transports such as NCCL require both endpoints to agree on buffer sizes before initiating a transfer.

We design and implement StreamInfer, an MoE decoding system that realizes StreamEP by addressing the aforementioned challenges with the following mechanisms: a *layer-wise adaptive token batching* LLM execution infrastructure (§4.3); a set of *batch reformation optimizations* that minimize kernel launch and metadata transfer overhead (§4.5); a *defragmenting scheduler* that consolidates partial batches into efficient executions while guaranteeing forward progress (§4.4); and a *two-phase communicator* that supports fully asynchronous, variable-size GPU-to-GPU transfers over vanilla NCCL without IBGDA (§4.6).

StreamInfer targets throughput-optimized serving: batch inference workloads such as synthetic data generation, offline evaluation, and bulk summarization, as well as high-load online serving where maximizing tokens per second from a fixed GPU budget is the primary objective. We evaluate StreamInfer on GPT-OSS-120B and GLM-4.5-Air with ShareGPT and GSM8k workloads using a 16×L40S cluster and a 16×A100 cluster, both with 200 Gbps inter-node networking. StreamInfer achieves up to 1.7× higher decoding throughput compared to traditional EP and maintains near-constant throughput under realistic network interference that degrades EP by up to 24% (§6.5), demonstrating that efficient large-scale MoE serving is achievable on commodity accelerators and fabrics.

2 Background

2.1 Why Expert Parallelism for MoE Decoding

Serving a large MoE model whose parameters far exceed the memory of a single GPU requires distributing the model across multiple devices, often spanning multiple nodes. Three parallelism strategies are available: pipeline parallelism (PP), tensor parallelism (TP), and expert parallelism (EP). We argue that EP is the only viable choice for MoE *decoding*, while PP remains effective for prefill. This mismatch makes prefill–decode disaggregation a near-necessity for efficient large MoE serving.

Pipeline parallelism is ineffective for MoE decoding. PP partitions the model into N sequential stages and splits the global batch of B tokens into N micro-batches of B/N tokens each [14, 28]. For dense model training with large batches, this yields efficient pipeline utilization. For MoE decoding, however, PP is counterproductive. Autoregressive decoding is memory-bandwidth bound: each token requires reading the full expert weights from GPU memory, so throughput depends on *arithmetic intensity*, the ratio of computation to memory traffic [44]. MoE routing is already sparse; each token activates only k experts, so the effective per-expert batch size is small to begin with. PP further divides this already-small batch by N, causing arithmetic intensity to collapse when the cluster size is large, while extra problems like pipeline bubbles are introduced.

Tensor parallelism does not scale beyond a single node. TP shards weight matrices across devices and distributes the computation of each layer, preserving arithmetic intensity since the full token batch participates in every shard's computation [38]. However, TP requires all-reduce collectives after *both* the attention and the MLP sub-layers of each transformer block, demanding high bisection bandwidth across all participating devices. This restricts TP to devices connected by intra-node interconnects such as NVLink or NVSwitch [38]. Fitting a large MoE model entirely within a single node demands HGX-class machines with 8 high-VRAM GPUs, hardware that is scarce in academic and many enterprise settings (Table 2). Moreover, as the TP degree increases, per-device tensor slices shrink, reducing kernel efficiency [38].

EP is the natural fit for multi-node MoE decoding. EP sidesteps both limitations mentioned above. By assigning a subset of experts to each device, EP preserves high arithmetic intensity: each device processes all tokens routed to its experts using full-size weight matrices, keeping the compute-to-memory ratio favorable even during decoding [21]. This advantage usually grows when the EP-domain is larger. When paired with data-parallel attention (replicated attention weights, no cross-device communication), collectives are confined to the MoE sub-layers, with two per MoE layer: dispatch and combine. These collectives are implemented either as all-to-all exchanges [4, 48] or as an all-gather of tokens followed by an all-reduce of expert outputs [49]. This eliminates the per-block attention all-reduces that TP would require. EP also maintains high kernel efficiency, since each device executes standard-sized matrix operations on its local experts. These properties have made EP the de facto standard for multi-node MoE

Table 1. Hardware requirements of MoE-optimized EP communication stacks.

Library	Overlap	GPU	Network
DeepEP LL [48]	Two batch overlap	SM90+	GPUDirect RDMA
pplx-kernels [23]	Two batch overlap	SM90+	GPUDirect RDMA
fabric-lib [24]	Two batch overlap	SM90+	GPUDirect RDMA
UEP [26]	Two batch overlap	SM90+	GPUDirect RDMA
NCCL EP [11]	Two batch overlap	SM90+	GPUDirect RDMA
FlashMoE [1]	Fused	SM70+	NVLink
Comet [47]	Fused	SM90+*	NVLink
NCCL (vanilla)	—	Any	Any

SM90+ = Hopper or newer architectures. *Comet's fused communication–computation kernels use SM90-specific features (TMA, WGMMA).

serving [4, 48].

PP for prefill and the case of disaggregation. While PP fails for decoding, it is well-suited for prefill, where long input sequences are processed in a single forward pass. Prefill is compute-bound: hundreds or thousands of tokens per request provide ample arithmetic intensity even after micro-batch splitting. PP's low communication overhead (activations transferred only at stage boundaries) is a genuine advantage over EP/TP's per-layer collective communications. This phase-dependent optimality gap motivates *prefill–decode disaggregation* [31, 32, 50], where prefill and decode run on separate device pools each using its best-fit parallelism. For large MoE models on commodity clusters, disaggregation is not merely an optimization but approaches a necessity: PP enables prefill to scale across nodes with minimal communication, while EP keeps decoding efficient despite sparse routing. This paper focuses on the decode side of this split.

2.2 Optimizations on MoE

EP's dispatch and combine collectives are the dominant latency source during MoE decoding, and the systems community has responded with increasingly specialized communication stacks, most built around all-to-all token exchanges. We review two main approaches, two-batch overlapping and fine-grained kernel fusion, and examine both their overlap potential and their deployment requirements.

Two-batch overlapping (TBO). The most widely adopted overlap strategy, introduced by DeepSeek-V3 [4], splits each iteration's token batch into two micro-batches and interleaves them: while one micro-batch performs expert computation, the other executes its all-to-all dispatch or combine. SGLang [49] and vLLM [19] have adopted this approach for MoE serving. Effective TBO does not require a strictly zero-SM communication backend. It requires communication and expert kernels to make concurrent progress with limited interference. Vanilla NCCL can run communication on a separate stream and could be extended specifically for TBO. But even under ideal overlap, TBO preserves the collective barrier for each micro-batch, so the slower of communication and computation remains exposed.

Table 2. Freely available GPU resources across major U.S. academic HPC clusters (no condo/PI-purchased hardware).

Cluster	H+	Pre-H	HGX	Fabric
National (ACCESS-allocated)				
Delta GPU	64 (7%)	848	64 (7%)	SS 200G
DeltaAI	528 (100%)	–	–	SS 200G
Bridges-2 GPU	80 (22%)	288	80 (22%)	IB 400G
Anvil GPU/AI	84 (57%)	64	–	IB 100–200G
Derecho GPU	–	328	–	SS 200G
Jetstream2 GPU	–	360	–	Eth 100G
Others	16 (6%)	240	–	Mixed
Campus shared (institution-wide, free access)				
MIT Engaging	68 (26%)	196	64 (24%)	IB NDR
MIT SuperCloud	–	488	–	–
Harvard Cannon	–	208	–	IB HDR
USC Discovery	–	70	–	IB 100–200G
Stanford Sherlock	–	~75	–*	IB NDR
Berkeley Savio	–	~40	–	IB FDR
Total	**840 (21%)**	**3,205**	**208 (5.1%)**	

H+ = Hopper+ (H100/H200/GH200); percentages are H+ share of each cluster's total. Pre-H = A100/A40/L40S/V100/T4/MI100. HGX = Hopper+ GPUs in ≥8-GPU-per-node configs; percentages are HGX share of cluster total. SS = HPE Slingshot. IB = InfiniBand. *Condo-owners partition only; excluded from HGX count.

Let b denote the number of tokens in one micro-batch, and let C and M denote its computation and communication time. Without overlap, the micro-batch throughput is $T = b/(C + M)$. Each micro-batch still occupies the compute resource for C time and the network for M time, giving the steady-state bound

$$T_{\text{TBO}}^{\text{ideal}} \leq \frac{b}{\max(C, M)} = T\frac{C + M}{\max(C, M)}. \quad (1)$$

For communication-bound execution, where $M > C$, the maximum speedup is $(C + M)/M = 1 + C/M$. The benefit therefore shrinks as communication becomes more dominant. Pipeline fill and drain, kernel interference, and lower micro-batch kernel efficiency reduce the attainable speedup further. We compare StreamInfer against an optimistic TBO projection based on this bound; StreamInfer still achieves higher throughput (§6.2).

Fine-grained kernel fusion. Rather than overlapping two separate kernels on different CUDA streams as TBO does, Comet [47] and FlashMoE [1] fuse communication and computation *within a single GPU kernel* at tile granularity (labeled "Fused" in Table 1). Specialized thread blocks pull tokens from remote GPUs via NVLink while other thread blocks execute expert GEMMs on already-arrived tiles, overlapping both dispatch and combine with computation at a much finer grain than micro-batch alternation. However, these fused kernels are tuned for NVLink-class low-latency interconnects. Their fine-grained per-token remote accesses stall on higher-latency links such as IB or RoCE, so they lose generality on commodity clusters.

The hardware gap. Most optimized NVIDIA paths in Table 1 require Hopper-class GPUs. DeepEP [48], UEP [26],

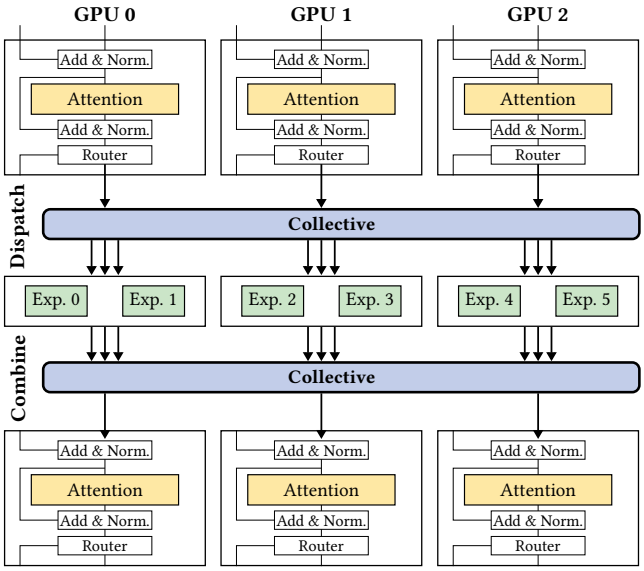

Figure 3. An MoE transformer layer: each token is routed to a subset of experts by a learned gating function. Expert parallelism distributes experts across devices; tokens are exchanged via barrier-style collectives before and after expert computation. Every device must wait at the barrier before executing the next layer.

NCCL EP [11], and Comet [47] use SM90 mechanisms such as the Tensor Memory Accelerator (TMA) to move token tiles asynchronously while preserving SM resources for expert computation. fabric-lib [24] avoids these SM90-specific mechanisms, but ships kernels only for SM90 and newer GPUs. FlashMoE [1] supports pre-Hopper GPUs, but its fused design requires the NVLink fabric to stay efficient. The relevant requirement is therefore the combination of a recent GPU and a compatible high-bandwidth, low-latency GPU network. Only 21% of surveyed academic GPUs are Hopper or later, and just 5.1% reside in HGX-class nodes (Table 2). Commercial cloud server support is similarly limited to specific instance and backend combinations. AWS EFA supports UEP [26] and fabric-lib [24] only on P5. GCP support ranges from no direct GPU networking on A2 to specialized GPUDirect paths on A3 and A4. Vanilla NCCL therefore remains the broadest common interface across these academic and cloud GPU clusters for commodity use.

The barrier remains. These specialized mechanisms reduce communication overhead and overlap transfers with expert computation, but they retain operator-level batch synchronization. TBO hides only the communication matched by concurrent computation. Fused operators such as FlashMoE pipeline communication and computation at tile granularity, but the batch cannot advance until all required transfers complete. A slow transfer therefore delays global progress in both designs. StreamInfer operates under the same network-bandwidth constraints, but removes this global advancement barrier, allowing each GPU to process ready tokens while slower transfers continue (§3).

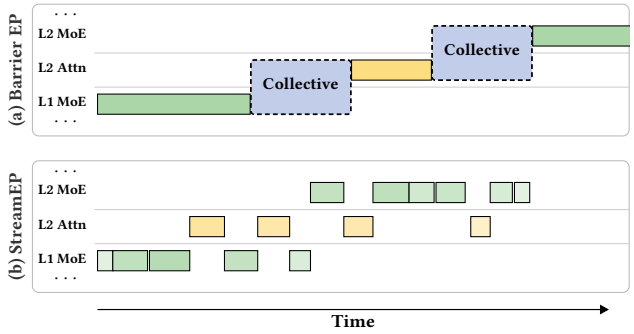

Figure 4. Demonstration of a single rank's execution timeline for three consecutive layers under (a) barrier EP and (b) StreamEP.

3 From Barrier EP to StreamEP

As illustrated in Figure 3, EP distributes experts across devices and uses a pair of collectives, dispatch and combine, at every MoE layer to shuffle tokens to the devices hosting their assigned experts and return results. These collectives are barrier-style: every GPU must wait for the *entire* token batch to arrive before it can begin executing a layer.

Stream Expert Parallelism (StreamEP) takes a different approach: it removes the barrier entirely. Instead of assembling a complete batch before executing a layer, each GPU begins computation as soon as *any* tokens arrive and forwards results immediately, without waiting for the rest of the batch (Figure 1(b)). This applies uniformly across the transformer: expert dispatch and combine, attention, and any other operation that follows a collective exchange all proceed in a streaming fashion. Fast intra-node transfers deliver tokens first; the GPU computes on them while slower inter-node transfers complete in the background. There is no global synchronization point; each GPU's execution is driven purely by token arrival.

Removing the barrier turns the straggler impact from global to local. Under barrier EP, a single bottleneck straggler link stalls every other GPU at the collective barrier (Figure 4, top). Under StreamEP, a straggler only delays the tokens it carries; every other GPU processes its own tokens unimpeded (Figure 4, bottom). As a result, latency improves because the worst-case straggler no longer sets the pace for the entire system.

A natural concern is that eliminating the barrier will fragment token batches into many small, arithmetically inefficient executions, resulting in low throughput. The reality is the opposite. Under barrier EP, the GPU idles after early-arriving tokens are ready, waiting for the remaining tokens to complete their transfers. Under StreamEP, that same idle interval becomes useful computation (Figure 4): the GPU processes early arrivals immediately, and later tokens queue during execution, arriving to form the next batch without any explicit waiting. The spread in arrival times that was previously a source of idle time is now the mechanism that *creates* batching. Crucially, this batching is self-regulating.

When computation is the bottleneck, tokens naturally queue while the GPU processes earlier arrivals, accumulating into batches large enough to attain higher arithmetic intensity. When communication is the bottleneck, the GPU processes smaller batches sooner, reducing per-step latency rather than idling at a barrier. This matters because MoE decoding cannot increase batch size arbitrarily: KV cache memory caps the number of concurrent requests. Throughput, therefore, improves by reducing per-step latency.

StreamEP also differs fundamentally from existing overlap techniques. Two-batch overlapping (TBO) [4] and tile-level kernel fusion [47] reduce the *visible* cost of the collective by overlapping communication with computation from a concurrent micro-batch or tile. However, neither technique removes the barrier, and optimized implementations have limited hardware compatibility: a straggler still gates collective completion. StreamEP instead uses point-to-point transfers to forward each token group as it becomes ready and begins execution when that group arrives.

Realizing StreamEP, however, introduces system challenges. Without a collective to anchor layer boundaries, the system must manage asynchronous token arrival, batch reconstruction, and dynamic communication efficiently. §4 presents the design of StreamInfer, which addresses each of these concerns.

4 StreamInfer Design

StreamEP's barrier-free execution introduces several novel system challenges: each GPU must independently manage tokens arriving at unpredictable times, reconstruct efficient batches from scattered arrivals, and communicate variable-size payloads without collective synchronization. We address these in StreamInfer, an MoE decoding system that implements StreamEP efficiently.

4.1 System Overview

Figure 5 shows the architecture of StreamInfer. The system comprises a *global coordinator* and per-GPU *StreamEP engines*.

Global coordinator. The coordinator is a CPU-based service that manages request lifecycle and cluster state. It consists of three components: (1) an API server that accepts serving requests, maintains request states throughout auto-regressive decoding, and handles tokenization/de-tokenization; (2) an attention load balancer that considers both active request count and KV cache occupancy to select the least-loaded data-parallel (DP) rank for each new request. Once assigned, a request remains bound to the same DP rank throughout its decoding process to enable KV cache reuse; and (3) a cluster manager that initializes communication channels between engines and tracks GPU memory usage for the load balancer.

StreamEP engine. Each GPU has a dedicated StreamEP engine that hosts both attention layers (replicated under DP) and expert layers (partitioned under EP), consistent with standard MoE serving systems [19, 49]. The critical difference is that StreamInfer does not use blocking collectives for dispatching and combining tokens across parallel GPUs. Instead, each engine independently receives tokens from any peer, executes the appropriate layer (either attention or expert), and forwards results to the next destination, all without global coordination.

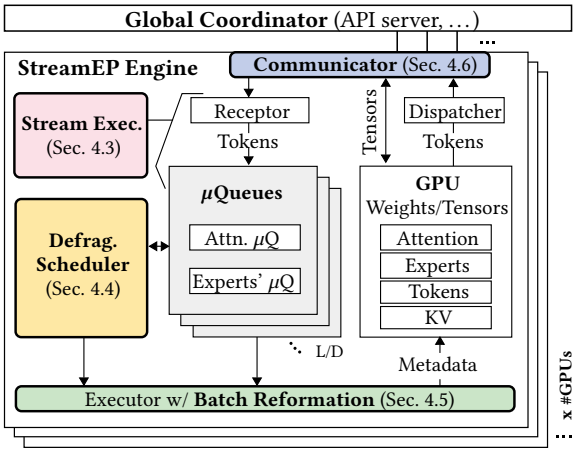

Figure 5. System architecture of StreamInfer. Each GPU runs a StreamEP engine with μ-queues, a defragmenting scheduler, an executor, and a communicator. The global coordinator manages requests and load balancing across engines.

The engine design is separated into a *data plane* and a *control plane*: each token's tensor data remains in GPU memory throughout its lifetime and is transferred directly between GPUs via NCCL, while the engine tracks and routes these GPU-resident tensors through lightweight CPU-side metadata (§4.2). This avoids unnecessary host–device copies of token tensor data and allows the CPU to make scheduling and routing decisions without touching the GPU data path. Based on this principle, each engine is organized around four components that map directly to the challenges:

- A **receptor and μ-queues** that manage tokens arriving asynchronously from multiple sources, sorting them into per-layer queues via metadata, addressing *layer-level asynchronous execution* (§4.3).
- A **defragmenting scheduler** that selects which layer to execute next, consolidating fragmented token batches into efficient executions while guaranteeing forward progress, addressing *batch fragmentation* (§4.4).
- An **executor** with **batch reformation optimizations** that efficiently reconstruct contiguous batches from individually arrived tokens while overlapping CPU-side metadata preparation with GPU computation, addressing *batch reformation overhead* (§4.5).
- A **two-phase communicator** that supports fully asynchronous, variable-size GPU-to-GPU transfers over NCCL P2P without requiring IBGDA or NVSHMEM, addressing *dynamic point-to-point communication* (§4.6).

Executor Metadata for Tokens

- **`RequestID`**: tracks generated output token and attention DP rank
- **`LayerID`**: indicates the layer this token should be used as input for `<layer#> | <attn. DP rank> | <expert#>`
- **`Tensor`**: reference to the input tensor on the GPU
- **`Topk_weights`**: used for top-k token merging

Figure 6. Token metadata tracked on the CPU. `LayerID` encodes the target layer as `<layer#> | <attn. DP rank> | <expert#>`.

The receptor, dispatcher, and communicator each run in dedicated CPU threads, decoupled from the main execution thread that drives the scheduler and executor. This allows token arrival and departure to proceed without blocking GPU computation.

Shared experts. Some MoE architectures include *shared experts* that process every token at each layer, in addition to the top-K routed experts [4]. Since shared experts are usually a small fraction of the entire MoE layer and every token passes through shared experts, StreamInfer executes them under data parallelism, immediately after the attention computation on the same GPU, before dispatching token copies to the routed experts.

4.2 Token Metadata and μ-Queues

Without collective synchronization, each GPU must independently manage tokens arriving at unpredictable times from multiple peers, determine which layer each token needs, and decide the execution order, none of which is globally coordinated. The execution engine handles this by associating each token's GPU-resident tensor data with CPU-side metadata that tracks its identity and destination.

We consider an MoE model with L layers, where each layer comprises attention computation followed by expert computation. We group gating and top-K merge with the attention stage, since both execute locally on the token's home DP rank. Tokens within each engine are organized into per-layer *μ-queues*: each layer has two μ-queues, one *attention queue* for tokens awaiting that layer's attention and one *expert queue* for tokens awaiting that layer's expert computation. This separation reflects the two distinct routing patterns in MoE: attention is executed locally on the token's home DP rank, while expert computation may require forwarding to a remote GPU. The engine thus maintains $2 \times L$ μ-queues in total.

Figure 6 lists the metadata associated with each token. Since asynchronous queuing may reorder tokens arbitrarily, position-based identification is no longer possible, so each token carries a `RequestID` to preserve its identity. The key routing field is `LayerID`, a three-field tuple that the communicator uses to determine the destination GPU and the receptor uses to select the correct μ-queue. Its `attn. DP rank` field is preserved throughout a token's lifetime to ensure it always returns to the GPU holding its KV cache, while `expert#` is set to `null` when the token is attention-bound and its expert assignment has not yet been determined.

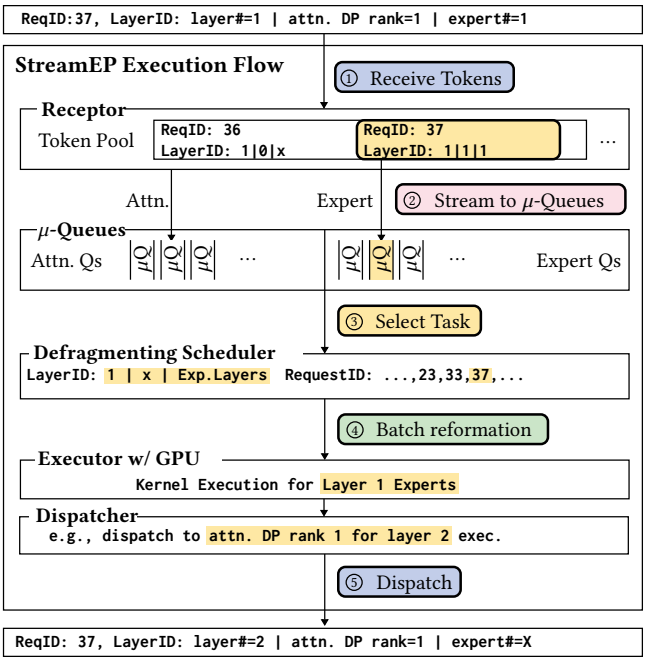

Figure 7. Token execution flow within a StreamEP engine (with request 37 as an example). ① Receive dispatched tokens from other GPUs. ② Stream to layer-specific μ-queues (attention or expert) based on `LayerID`. ③ Select the optimal task via the defragmenting scheduler. ④ Perform batch reformation and launch GPU kernels. ⑤ Relabel output tokens with next-layer destinations and forward.

4.3 Execution Flow of StreamEP Engine

Figure 7 illustrates the five-stage token pipeline within each engine. We explain each stage using request 37 as a running example: a token targeting expert 1 at layer 1, originally from attention DP rank 1. Each token carries a compact `LayerID` that encodes this routing information; the full metadata specification is in §4.2.

Stage ①: Receive tokens. The receptor is the entry point for token batches arriving from other engines. The communicator (§4.6) transfers tensor data directly into local GPU memory and hands the batch to the receptor, which processes only the accompanying CPU-side metadata for routing. In the running example, request 37 has been dispatched to the local engine from another GPU after completing attention at layer 1. The receptor places incoming tokens into a *token pool*, where they are held alongside other recently arrived tokens.

Stage ②: Stream to μ-queues. From the token pool, the receptor inspects each token's `LayerID` and routes it to the appropriate μ-queue. As described above, each layer maintains an attention μ-queue and an expert μ-queue. Request 37, whose `expert#=1` specifies a particular expert, is routed to the expert μ-queue for layer 1. Request 36, with `expert#=null`, would instead be routed to the attention μ-queue for layer 1. This is where StreamEP's natural batching

takes effect: while the GPU is busy executing another layer, tokens from different sources accumulate in the μ-queues, coalescing into batches large enough for efficient computation without any explicit synchronization.

Stage ③: Select task with scheduler. Whenever the GPU becomes idle, the defragmenting scheduler (§4.4) examines all μ-queues and selects the best (layer, type) combination to execute next. Continuing the example, the scheduler selects layer 1's experts, which have accumulated tokens from requests 23, 33, 37, and others. The selected μ-queue's tokens are then passed to the executor for batch reformation and computation.

Stage ④: Batch reformation and execution. Since tokens arrive individually from different GPUs, the executor must first *reform* them into a contiguous batch before launching computation (§4.5). It then launches the GPU kernels for layer 1's expert computation.

Stage ⑤: Dispatch. After expert execution completes, the dispatcher determines each output token's next destination. For request 37, which just finished expert processing at layer 1, the next step is the attention layer of layer 2. The dispatcher relabels the token as `LayerID: layer#=2`, `attn. DP rank=1`, `expert#=null`. This relabeling advances the token to layer 2, preserves attention DP rank 1 so the token returns to the GPU holding its KV cache, and sets `expert#=null` because the expert assignment for layer 2 will only be determined by the router after layer 2's attention computation. If the target GPU is remote, the dispatcher forwards the token via the communicator (§4.6); if the target attention DP rank is hosted locally, it is enqueued directly into the local attention μ-queue.

Top-K routing. MoE models route each token to K experts per layer; the final output is the weighted sum of all K expert results. The five-stage flow naturally accommodates this: in Stage ⑤, the dispatcher sends one copy to each of the token's K assigned experts, and the results return independently. Since the K copies may arrive at different times, the receptor tracks partial arrivals in the token pool and promotes a token to its μ-queue only once all copies have been received; we detail the reassembly and merge mechanism in §4.5.

4.4 Defragmenting Scheduler

Removing the barrier means that token batches are no longer delivered whole: tokens trickle in from different GPUs at different times, and the same logical batch may fragment across multiple μ-queues. While StreamEP's μ-queuing mitigates this by letting tokens accumulate while the GPU is busy, the *scheduling policy* determines how effectively these fragments consolidate. We find that the policy significantly impacts both latency and throughput.

Strawman 1: Most-token-first-serve (MTFS). A natural strategy is to always execute the layer with the most queued tokens, maximizing immediate batch size. However, MTFS causes severe *batch fragmentation* (Figure 8(a)). Without blocking collectives that merge tokens from all sources, token batches naturally arrive in pieces, as some GPUs return their outputs before others. MTFS exacerbates this by tending to leave the last slice of tokens for each layer unexecuted, since another layer may have accumulated more tokens by then. These orphaned fragments cascade through the timeline, resulting in many layers with scattered small batches rather than a few layers with consolidated large batches.

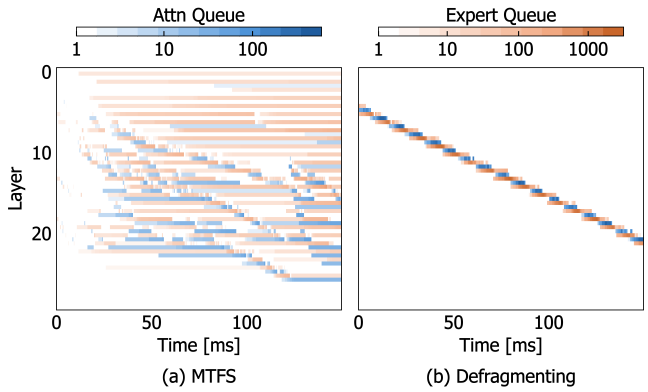

Figure 8. μ-queue depth over time under the two scheduling policies. (a) MTFS scatters tokens across many layers simultaneously, producing fragmented small-batch executions. (b) StreamInfer's defragmenting scheduler keeps execution concentrated in a single frontier wave that sweeps the layers. For visibility, we only show a subset of layers.

Strawman 2: First-layer-first-serve (FLFS). The opposite extreme strictly prioritizes earlier layers (lower layer number), aggressively defragmenting by funneling all tokens into a single frontier wave. FLFS performs reasonably well because autoregressive decoding naturally reinforces consolidated waves across iterations: a well-merged wave in one iteration benefits the next. However, it has a critical flaw: newly arriving requests enter at the earliest layer and always take priority, potentially causing livelock under high request rates as the system perpetually processes new requests without advancing existing tokens to completion.

StreamInfer's defragmenting scheduler. Our scheduler balances the strengths of both strawmen: it promotes defragmentation like FLFS while considering queue occupancy like MTFS, avoiding both excessive fragmentation and livelock. Algorithm 1 presents a simplified version. For each (layer, type) pair, the scheduler computes a score by combining the current queue depth with a weighted *lookahead score*. The lookahead walks the execution chain (attention, expert, next layer's attention, and so on) for W steps, summing estimated occupancy with exponentially decaying weights (controlled by decay factor δ). Crucially, the occupancy estimate at each lookahead step combines the current queue depth with recently drained (executed) token counts from the last B scheduling steps. This prevents a just-drained layer from appearing cold, because other ranks are likely still processing

Algorithm 1 Defragmenting Scheduler

Input: N_L: #layers, $Q[l,t]$: tokens in layer l's queue ($t \in \{\text{attn}, \text{expert}\}$), δ: weight decay, W: lookahead window, B: history window, $D_i[l,t]$: tokens drained i steps ago ($1 \le i \le B$)
Output: (l^*, t^*): optimal (layer, type) to schedule
Init. $Scores[N_L][2] \leftarrow 0$
function NEXT(l, t) ▷ attn→expert→next layer's attn
  return (l, expert) if $t = \text{attn}$, else $(l{+}1, \text{attn})$
end function
for $l \leftarrow 0$ to $N_L - 1$ **do**
  for each $t \in \{\text{attn}, \text{expert}\}$ **do**
    if $Q[l][t] > 0$ **then**
      $LScore \leftarrow 0$
      $(l', t') \leftarrow \text{Next}(l, t)$
      for $k \leftarrow 1$ to W **do**
        if $l' < N_L$ **then**
          $Occ \leftarrow Q[l'][t'] + \sum_{i=1}^{B} D_i[l'][t']$
          $LScore \leftarrow LScore + Occ \times \delta^k$
          $(l', t') \leftarrow \text{Next}(l', t')$
        end if
      end for
      $Scores[l][t] \leftarrow Q[l][t] + LScore$
    end if
  end for
end for
$(l^*, t^*) \leftarrow \arg\max_{l,t} Scores[l][t]$ ▷ Pick the best (layer, type)
return (l^*, t^*)

tokens nearby and new arrivals are in flight. The scheduler therefore keeps execution concentrated around the active wave rather than scattering to unrelated layers (Figure 8(b)). Unlike FLFS, the scheduler can defer small fragments that are far from the main wave and whose execution would not aid defragmentation, allowing StreamInfer to tolerate occasional severe stragglers; unlike MTFS, it resists fragmenting tokens across many layers simultaneously.

4.5 Batch Reformation

In traditional EP, the dispatch and combine collectives deliver tokens in a fixed order that remains contiguous across layers, so no per-layer batch reconstruction is needed. StreamEP breaks this invariant: tokens arrive asynchronously from different GPUs and land at scattered GPU memory locations.

Before launching a layer's computation, the executor must *reform* these tokens into a contiguous batch. After the scheduler selects a μ-queue, the CPU collects the `Tensor` references of all queued tokens and transfers them to the GPU. A reformation kernel then gathers the scattered token tensors into a single contiguous input buffer.

For attention layers, the CPU must additionally collect per-token context lengths and look up each token's KV cache page table indices under paged attention [19], since tokens from different requests map to different physical pages.

Top-K reassembly. When top-K > 1, each token's K expert copies travel independently and may arrive at different times. The receptor tracks partially reassembled tokens in the token pool, keyed by `<RequestID, LayerID>`: each arriving

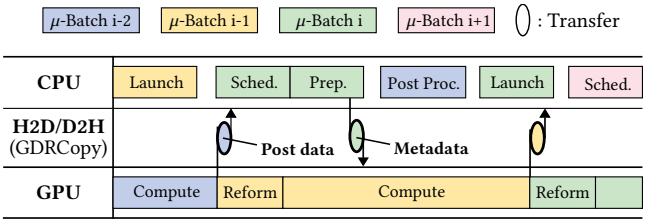

Figure 9. Batch reformation overhead hiding. While the GPU computes μ-Batch $i{-}1$, the CPU schedules and prepares μ-Batch i, and GDRCopy transfers post-processing data for μ-Batch $i{-}2$ and metadata for μ-Batch i in parallel.

copy adds its tensor reference and gating weight to the corresponding entry. Once all K copies have arrived, the receptor promotes the token to the appropriate attention μ-queue. The weighted merge is then performed during batch reformation: for attention layers following top-K expert computation, the reformation kernel fuses the gather with the weighted merge of each token's K copies, producing one merged tensor per token. This amortizes kernel launch overhead by merging all tokens in the scheduled batch in a single pass; each token's `Tensor` metadata is updated to reference the merged result.

Overlapping reformation with computation. Although reformation is a new per-layer cost, it can be mostly hidden. While the GPU computes μ-Batch $i{-}1$, the CPU concurrently schedules and prepares μ-Batch i, transferring its metadata to the GPU (Figure 9). Once the GPU finishes μ-Batch $i{-}1$, it immediately runs the reformation kernel for μ-Batch i and launches computation without stalling for CPU-side work. Post-execution results such as gating indices and expert assignments are similarly copied back to the CPU for the dispatcher. All metadata transfers use GDRCopy [37], which minimizes host–device copy overhead; we detail this in §5.

4.6 Two-Phase Communicator

As the background showed (§2.2), all existing MoE-optimized communication stacks require hardware capabilities such as IBGDA and InfiniBand that might be unavailable on commodity clusters. The only universally available communication library is NCCL, but its point-to-point APIs pose two challenges for asynchronous operation: (1) both sender and receiver must invoke `NCCLSend`/`NCCLRecv` simultaneously to initiate a transfer, and (2) the receiver must know the sender's rank and tensor size in advance. In StreamInfer, due to asynchronous scheduling, any engine may receive variable-size batches from any other GPU at any time. The communication pattern is no longer the structured collective exchange that barrier EP provides.

Figure 10 illustrates our solution: a two-phase communication protocol that enables fully asynchronous, variable-size transfers over NCCL without any specialized hardware.

Phase 1: Metadata exchange. Before initiating GPU-side data transfer, the sender transmits token metadata including

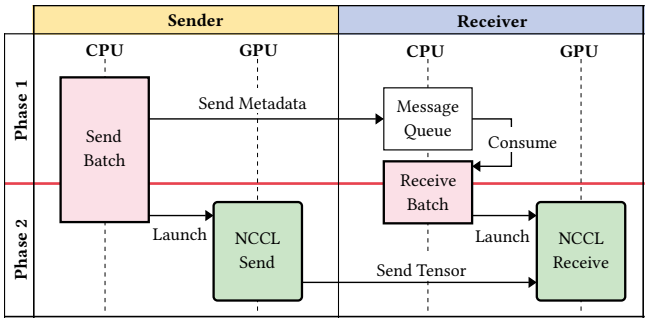

Figure 10. Two-phase communication protocol. Phase 1 exchanges lightweight metadata via the message queue on the CPU to coordinate buffer allocation. Phase 2 transfers tensor data via NCCL on the GPU.

source rank, number of tensors, and tensor sizes to the receiver through lightweight CPU-side message queues [13]. Each communicator maintains a message queue that iteratively consumes metadata from all senders. Upon receiving metadata, the receiver allocates an appropriately sized NCCL receive buffer, resolving the buffer-size agreement problem. For example, when the dispatcher forwards request 37's output (now labeled `layer#=2`, `attn. DP rank=1`, `expert#=null`) to the GPU hosting attention DP rank 1, it first sends the token metadata via the message queue so the receiver can prepare the corresponding buffer.

Phase 2: Tensor transfer. With buffer sizes established, both sides launch their respective `NCCLSend` and `NCCLRecv` kernels on dedicated GPU streams. Critically, after launching the NCCL kernels, the CPU proceeds immediately to the next communication task that checks the message queue for new metadata and coordinates another transfer without waiting for the GPU transfer to complete. The receiver synchronizes with the GPU only before releasing the received tensor to the receptor, ensuring data integrity. The sender defers synchronization: it checks for completed sends only to reclaim send buffers and throttles concurrent NCCL sends to bound memory usage. This design, combined with dedicated send and receive threads that avoid distributed deadlock, allows the communicator to manage multiple concurrent transfers across all peers efficiently, achieving the dynamic point-to-point communication that StreamEP requires without any hardware beyond NCCL support.

5 Implementation

We implement StreamInfer from scratch, adopting model layer implementations from vLLM [19] and SGLang [49]. The model executor is written in Python, while the performance-critical components (communicator, receptor, scheduler, and dispatcher) are implemented with about 9K LoC in C++ and exposed via pybind11. The scheduler and executor run in the main Python thread; the receptor and dispatcher operate on dedicated POSIX threads with their own CUDA streams, enabling communication–computation overlap and bypassing Python's Global Interpreter Lock (GIL) for full concurrency.

Layer-wise CUDA graphs. We use CUDA graphs to amortize kernel launch overhead at low batch sizes. Unlike conventional systems that record G graphs for the entire model, StreamInfer operates at layer granularity, requiring $L \times G$ graphs total. To avoid the naïve memory footprint of per-graph static buffers, we share intermediate buffers across all layer graphs at the same batch size, reducing overhead by a factor of L.

Host–device coordination. The attention executor requires several fine-grained CPU–GPU transfers per layer: allocating KV slots, transferring per-token metadata to GPU for paged attention, and copying expert indices and gating weights back to CPU for token routing. To minimize latency of these small transfers, we employ GDRCopy [29, 37], which offers lower overhead than standard CUDA memcpy.

6 Evaluation

We evaluate StreamInfer on two commodity GPU clusters with two MoE models [9, 30] and two workloads [2, 3], comparing against four parallelism strategies in SGLang (§6.1).

Our evaluation answers the following questions:

- Does StreamInfer improve decoding throughput and inter-token latency over state-of-the-art parallelism strategies on resource-constrained clusters? (§6.2 and §6.3)
- How does StreamInfer scale with increasing request concurrency compared to barrier-based expert parallelism? (§6.4)
- Does StreamInfer tolerate network variability better than barrier-based expert parallelism? (§6.5)
- How much does each component of StreamInfer contribute to overall performance? (§6.6)

6.1 Evaluation Setup

Clusters. Our primary testbed, *Sphere*, comprises eight servers, each with two NVIDIA L40S GPUs (48 GB) connected via PCIe and a single ConnectX-6 NIC (200 Gbps RoCE) shared by both GPUs. We also evaluate on *Delta*, an academic cluster of four servers, each with four NVIDIA A100 GPUs (40 GB) connected via NVLink and a single 200 Gbps Cray Slingshot NIC per server. Both clusters represent commodity environments where per-GPU network bandwidth is limited and contended.

Models. We use two open-source MoE models in BFloat16 (Table 6). GPT-OSS-120B activates only 4.4% of parameters per token, making it highly communication-bound under EP; we use it as the primary model for all experiments. GLM-4.5-Air activates 11.3% via top-8 routing over smaller experts, producing denser activations and a larger per-token KV cache footprint; we use it in §6.4 to stress-test concurrency scaling under tighter memory budgets.

For details of both models, see §C. As MoE load-balancing is orthogonal to our work, we also apply EPLB-like expert

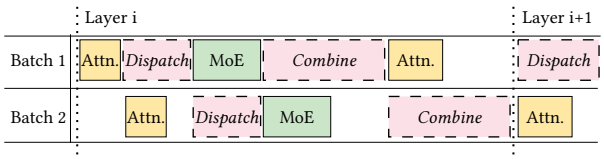

Figure 11. Two-batch overlap execution flow that interleaves communication and computation across two micro-batches.

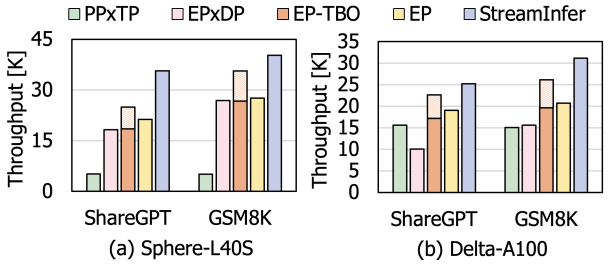

Figure 12. Decoding throughput comparison of StreamInfer on two resource-constrained clusters. For EP-TBO, the crosshatched segment represents the improvement attributed to projection.

placement to StreamInfer and the baselines. See §D for more details.

Baselines. We compare StreamInfer against four parallelism configurations in SGLang [49], the state of the art in open-source MoE serving:

- **Expert parallelism (EP)**: EP across all 16 GPUs, using all-gather for token dispatch and all-reduce for token combine at each MoE layer. This is SGLang's portable realization of the dispatch and combine collectives, and it carries the same barrier semantics as all-to-all-based EP.
- **Pipeline parallelism combined with tensor parallelism (PP×TP)**: PP avoids inter-node collectives but introduces pipeline bubbles and limits per-stage batch size. On Sphere we use PP×TP of degree 8×2; on Delta, 4×4 to keep TP intra-node.
- **Expert parallelism over two replicas (EP×DP)**: two model replicas, each serving with EP over half of the cluster.
- **Expert parallelism with ideal two-batch overlap (EP-TBO)**: EP with two micro-batches pipelined, overlapping one batch's communication with another's computation. Because the evaluated SGLang configuration does not support TBO on pre-Hopper GPUs, we measure EP at half of the maximum concurrency and apply the ideal bound in Equation (1), as illustrated in Figure 11.[1]

Beyond the main throughput comparison, we use EP as the sole baseline in subsequent experiments, as it is the best-performing baseline in practice.

[1]The half-concurrency run measures T for one TBO micro-batch. Our communication-bound setup gives $C/M \approx 0.33$, so the ideal factor is $1 + C/M \approx 1.33$.

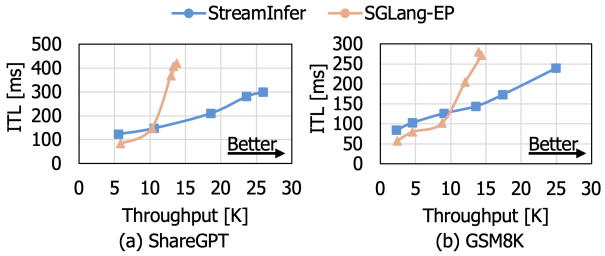

Figure 13. Throughput–ITL trade-off experiments.

Workloads and Metrics. We use ShareGPT [2], a chat workload with high length variance, and GSM8k [3], a math reasoning workload with shorter, more uniform lengths. Requests arrive via a Poisson process at a saturating rate with empirical length distributions. We report decoding throughput (tokens/s) and mean/P99 inter-token latency (ITL). For section 6.2, to measure the highest sustainable throughput, we use the metric from the peak 60-second window after the system reaches the highest global batch-size, excluding the ramp-up and long tails. For later sections, since we study the performance under different offered loads, we include metrics from the entire serving duration.

6.2 Throughput Analysis

Figure 12 compares decoding throughput for GPT-OSS-120B across both clusters at maximum sustainable concurrency.

Sphere (8×2×L40S). StreamInfer achieves 36 K tokens/s on ShareGPT and 40 K tokens/s on GSM8k, outperforming the best practical baseline (EP) by 1.7× and 1.4×, and the ideally projected EP-TBO by 1.4× and 1.1×, respectively. PP×TP delivers the lowest throughput at roughly 5 K tokens/s, as pipeline bubbles and small per-stage batch sizes severely underutilize the GPUs. EP reaches only 21 K and 28 K tokens/s on ShareGPT and GSM8k: 16-GPU collectives at every MoE layer are expensive over commodity networking, and the barrier forces all GPUs to wait for the straggler. EP-TBO overlaps communication with computation but halves the effective batch size, yielding 25 K and 35 K tokens/s.

Delta (4×4×A100). Delta differs from Sphere in three ways: smaller GPU memory (40 GB vs. 48 GB), NVLink instead of PCIe intra-node, and four GPUs per server sharing a single NIC. StreamInfer achieves 25 K and 31 K tokens/s on ShareGPT and GSM8k, outperforming EP by 1.3× and 1.5×. The relative advantage is smaller for two reasons. First, A100's smaller memory limits batch concurrency, reducing cross-node traffic and the communication bottleneck. Second, Delta's 4×4 topology places more GPUs behind fast NVLink, so only four inter-node NIC hops can gate a step. On Sphere, by contrast, all eight PCIe and eight NIC hops operate at comparable speeds and any can become a bottleneck. PP×TP reaches 15 K tokens/s, substantially higher than on Sphere, thanks to fewer pipeline stages and NVLink-accelerated tensor parallelism.

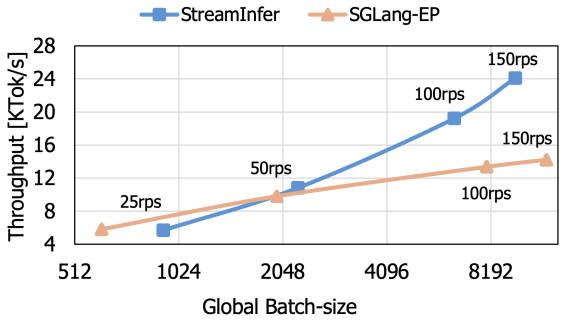

Figure 14. Decoding throughput against concurrency, the number of requests in flight, for GPT-OSS-120B on ShareGPT. Each point is annotated with the offered request rate that produced it.

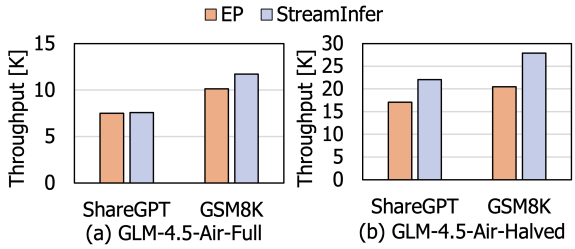

Figure 15. Throughput of StreamInfer and SGLang EP on GLM-4.5-Air with full and halved layers on Sphere.

EP-TBO over-estimates real performance. Recall that EP-TBO is an optimistic projection assuming zero communication-computation interference (§6.1); a real implementation would fall short, making StreamInfer's true advantage larger than reported.

Duplicating weights collapses concurrency. In EP×DP settings, each replica carries its own copy of the weights, leaving less VRAM for KV cache and shrinking the global batch, thus reducing concurrency and arithmetic intensity.

6.3 Throughput-Latency Tradeoff

We sweep the request rate and measure throughput and mean ITL simultaneously (Figure 13). EP plateaus at about 14 K tokens/s on both ShareGPT and GSM8k; beyond this point, higher request rates only inflate ITL. StreamInfer pushes the throughput frontier to 26 K and 25 K tokens/s. Under light load, StreamInfer exhibits slightly higher ITL due to queuing delay from asynchronous execution. As load increases, communication–computation overlap keeps StreamInfer's ITL growing slowly, while EP's lockstep barrier causes ITL to spike. With a moderately relaxed ITL SLO, StreamInfer delivers substantially higher throughput, reducing overall serving cost.

6.4 Concurrency Handling

In Figure 14, we plot throughput against different request rates to study the performance under different concurrency levels (i.e., global batch sizes). Notably, StreamInfer has a more significant throughput advantage when the concurrency is higher, because StreamInfer pays a fixed cost on every layer (e.g. scheduling, batch reformation, and metadata transfer), which is better amortized when each micro-batch is larger.

To further study the effect of concurrency on throughput, we switch to GLM-4.5-Air, whose attention head dimension (128 vs. 64 in GPT-OSS-120B) doubles per-token KV cache consumption, reducing maximum concurrency by over 50% under the same memory budget. We evaluate two configurations on Sphere: the full 46-layer GLM-4.5-Air and a halved-layer variant (23 layers) that frees memory to roughly double maximum concurrency, keeping all other variables (model architecture, routing, and cluster topology) fixed.

In a low concurrency scenario with a full model of 46 layers, KV cache limits concurrency, and StreamInfer achieves 7.6 K tokens/s on ShareGPT and 11.7 K tokens/s on GSM8k (Figure 15), a modest improvement of up to 1.2× over EP. At low concurrency, batches are small, leaving little room for asynchronous execution to accumulate tokens and amortize scheduling overhead. In a high concurrency scenario with a halved model with 23 layers, the freed memory roughly doubles maximum concurrency, and StreamInfer's advantage widens to 1.3× on ShareGPT and 1.4× on GSM8k. EP degrades under higher concurrency: larger batches amplify per-layer collective volume, and the barrier serializes on the slowest GPU at every layer. StreamInfer processes tokens without global synchronization, and higher concurrency naturally produces larger partial batches that improve GPU utilization rather than inflating communication cost. These results confirm that StreamInfer's throughput scales with concurrency while EP's barrier cost grows with it.

Table 3. Interference traffic scenarios used in the network variability experiment, ordered by increasing severity. Each scenario injects background traffic derived from the AWS trace in Figure 16(a).

Scenario	Description
Single	Background traffic on one inter-node link; remaining links unaffected
Single (2×)	Same as single-link with double the interference traffic volume
All	Four independent generators, one per inter-node link, covering all links simultaneously
Bidirectional-All (Bi-all)	Every node both sends and receives background traffic, exercising all links bidirectionally

6.5 Network Interference Tolerance

In both HPC and enterprise scenarios, commodity clusters can be shared among tenants, exposing serving workloads to unpredictable network interference, either as end-host NIC interference or transient network congestion. Alongside inference engines under 100 requests per second offered load, we replay an AWS interference trace [35] (Figure 16(a)) with a median bandwidth of 185.1 Gbps but a wide P1–P99 spread, and construct four scenarios of increasing severity (Table 3), from single-link to bidirectional all-link interference consuming only 7.5% of total bandwidth on average.

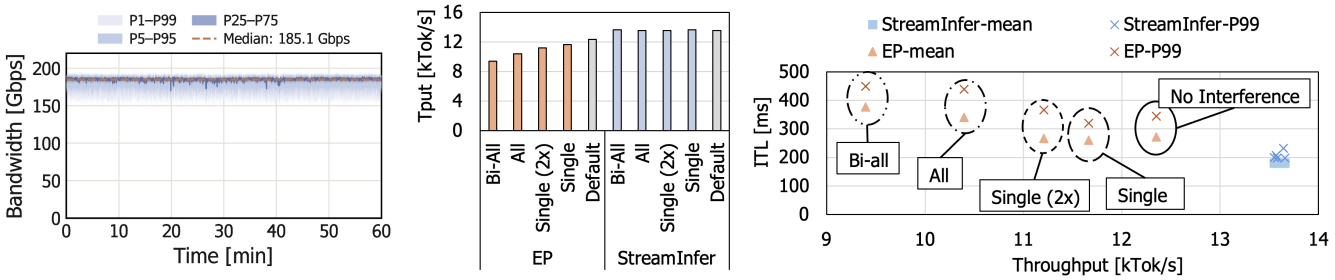

Figure 16. Network variability tolerance. (a) Bandwidth trace from an AWS cluster (median 185.1 Gbps) [35]. (b) Throughput comparison of SGLang EP and StreamInfer under progressively severe interference scenarios. (c) Throughput–ITL operating points under each scenario, showing that StreamInfer's operating point remains tightly clustered while SGLang EP shifts toward lower throughput and higher ITL.

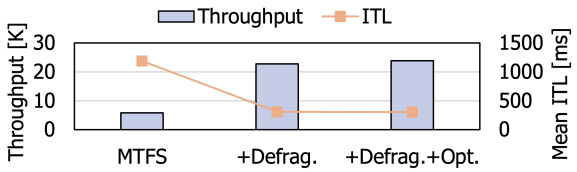

Figure 17. Ablation experiments of StreamInfer.

EP degrades as soon as a single link experiences interference (Figure 16(b)), losing up to ~24% throughput under bidirectional all-link interference, far out of proportion to the modest bandwidth consumed. The barrier serializes on the worst instantaneous link: the node with the highest spike gates the entire collective. Separating TCP control traffic from NCCL's RoCE datapath onto different NICs does not help, confirming the bottleneck is in the collective itself. StreamInfer maintains nearly constant throughput across all scenarios. Figure 16(c) shows EP's operating point drifting toward lower throughput and higher ITL as interference intensifies, while StreamInfer's operating points remain tightly clustered. Without a global barrier, localized interference delays only the tokens on affected links without propagating system-wide.

6.6 Ablation Study

We ablate StreamInfer's key components (Figure 17) using three configurations:

- **MTFS**: most-tokens-first-serve scheduling, no batch reformation optimizations, no CPU–GPU overlap.
- **+Defrag**: adds the defragmenting scheduler.
- **+Defrag+Opt** (full StreamInfer): adds batch reformation optimizations and CPU–GPU overlap.

The defragmenting scheduler is the largest contributor: switching from MTFS to +Defrag improves throughput by 3.2×. Without defragmentation, partial token batches are scheduled as many small executions, severely underutilizing GPU compute due to low arithmetic intensity. The scheduler consolidates fragments into larger batches, recovering arithmetic intensity. Batch reformation optimizations and CPU–GPU overlap (+Defrag+Opt) further improve throughput by 4.5% by reducing per-layer metadata transfer and kernel launch overhead, which accumulates rapidly as asynchronous scheduling increases the rate of layer invocations.

Table 4. Averaged per-batch CPU cost of the engine loop on GPT-OSS-120B, split by module, and the fraction of each stage hidden behind GPU execution. How well a batch overlaps depends on what batch precedes and follows it.

	Attention		Expert	
Stage	Per step	Hidden	Per step	Hidden
Schedule	127.2 μs	50.6%	29.6 μs	68.3%
Prepare	90.2 μs	50.0%	23.3 μs	66.9%
Launch	289.4 μs	66.6%	75.4 μs	65.9%
Post-process	126.6 μs	96.4%	57.8 μs	97.4%
CPU total	**633.4 μs**	**67.0%**	**186.1 μs**	**76.2%**
GPU execution	178.4 μs	—	437.3 μs	—

Table 4 further breaks down the CPU overheads of layer-wise scheduling and shows to what extent the overlapping techniques introduced in §4.5 can hide these overheads behind GPU computation. Example captured traces showing the CPU/GPU timelines are provided in §E. Notably, the percentage of hidden CPU work depends on the GPU kernel runtime of adjacent batches. Because an expert batch is usually surrounded by expert batches, and since expert batches have longer per-step compute time than attention, CPU-control overheads for experts are better overlapped. Also, scheduling and preparing an attention batch take more time than expert batches because there is more metadata associated with attention batches (e.g. KV-Cache management) to process.

6.7 End-to-End Serving

StreamInfer is a decode-only engine, so it needs to be deployed as the decoding worker of a prefill–decode disaggregation setup. Operating that architecture in production requires request routing and capacity planning to keep the prefill and decode throughput balanced under varying workloads, which is orthogonal to this work. For simplicity, we separately profile the SGLang prefill phase while offloading

Table 5. Profiled prefill time and projected makespan for serving a 5000-request batch on Sphere. In order to simulate the cost of KV-Cache transfer, the prefill time is measured while SGLang offloads resulting KV-Cache to Mooncake.

			Total makespan	
Workload	**In / Out**	**Prefill**	SGLang EP	StreamInfer
ShareGPT	174 / 435	12.4 s	171 s	**96 s**
GSM8k	137 / 287	8.8 s	111 s	**66 s**

to an external KV-cache service [32] and then use the decoding throughput measured in earlier sections to project the end-to-end serving time of request batches, each consisting of 5000 sequences sampled from the corresponding dataset.

As shown in Table 5, prefill takes only a tiny fraction of the end-to-end serving because it can process a whole prompt in one compute-bound forward pass, which is much more efficient than decoding. StreamInfer finishes the batch about 1.7× faster on both datasets. These end-to-end gains are close to its decoding advantage alone, exactly because prefilling is very fast under such workloads.

Overall, due to the fundamental throughput gap between prefill and decode, for decode-heavy and moderately-prefill-heavy workloads, StreamInfer can provide decent end-to-end speed-ups or reduce the required decoding hardware resources in a PD-disaggregated cluster. If the workload is extremely prefill-heavy or even prefill-only (e.g., LLM binary decision based on a long text), StreamInfer decoding will not be helpful.

7 Related Work

Mixture-of-Experts models. The sparse MoE layer [36] decouples model capacity from per-token compute by activating only a subset of experts per input. GShard [21], Switch Transformers [8], and GLaM [7] scaled this idea to trillions of parameters, and recent production models have all integrated MoE as the backbone, including Mixtral [16], DeepSeek [4, 6], Qwen [33, 45], Kimi [18], and GLM [10].

LLM and MoE serving systems. Orca [46] introduced continuous batching; vLLM [19] and SGLang [49] introduced fine-grained and structural KV cache management. DistServe [50], Splitwise [31], and Mooncake [32] disaggregate prefill and decode onto separate hardware. Attention-FFN disaggregation (AFD) [25, 40, 42, 51] enables flexible resource allocation across attention and expert modules. These optimize scheduling, memory, and phase-level allocation; StreamInfer targets a different bottleneck and dissolves the communication stall under EP in every MoE layer.

Communication optimization for distributed MoE deployment. A rich body of work reduces EP's communication cost (§2.2). FasterMoE [12] replicates hot experts locally; Tutel [15] dynamically switches collective patterns; Lina [22] optimizes expert placement and communication scheduling; DeepSpeed-MoE [34] provides end-to-end MoE training and inference with optimized primitives. More recent kernel-level fusion works, including DeepEP [48], pplx-kernels [23] and its successor fabric-lib [24], UEP [26], Comet [47], Flash-MoE [1], and MegaScale-MoE [17], streamline the all-to-all communication in EP and overlap it with expert computation. All retain the synchronization barrier, while StreamInfer eliminates it entirely, letting each GPU compute as tokens arrive. FAST [20] instead targets the communication load imbalance itself, scheduling all-to-all transfers so that each link carries a comparable volume. Its rebalancing happens inside each server before traffic crosses the network, so the approach relies on intra-node interconnects such as NVLink being far faster than inter-node links.

Parallelism strategies. Pipeline parallelism [14, 28] reduces arithmetic intensity during decode via micro-batch splitting; tensor parallelism [38] preserves batch size but requires per-layer all-reduce, limiting it to intra-node use; expert parallelism [21] maintains per-device intensity and is the de facto choice for multi-node MoE serving [4, 48]. Moebius [43] enables fast hot-switching between expert parallelism and tensor parallelism. StreamInfer keeps EP's placement model but replaces the synchronous dispatch and combine collectives with asynchronous point-to-point transfers, removing the barrier that makes EP vulnerable to stragglers.

Per-expert queuing. QLLM [39] introduces expert queues for batch-level preemption, while StreamEP uses the queues for dynamic re-batching.

8 Conclusion

This paper presented Stream Expert Parallelism (StreamEP), a paradigm that eliminates the collective synchronization barrier in MoE decoding by letting each GPU independently receive, compute, and forward tokens as they arrive. Our system, StreamInfer, realizes StreamEP on vanilla NCCL using per-layer μ-queues, a defragmenting scheduler, and a two-phase communicator. On 16×L40S and 16×A100 clusters, StreamInfer achieves up to 1.7× higher decoding throughput over traditional EP and maintains near-constant performance under network interference, demonstrating that the synchronization barrier, not the hardware, is the fundamental bottleneck for MoE serving.

Acknowledgments

We thank our shepherd and reviewers for their invaluable feedback. Development and evaluations used resources from SPHERE (funded by NSF #2330066); NCSA DeltaAI (funded by NSF OAC 2320345 and the State of Illinois); and USC's Center for Advanced Research Computing (CARC). This work was supported in part by a USC–Amazon Trusted AI research grant and by gifts and credits from Cisco, Google, and Amazon.

References

[1] Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. 2025. FlashMoE: Fast Distributed MoE in a Single Kernel. In *Proceedings of the Advances in Neural Information Processing Systems 38 (NeurIPS)*.

[2] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. https://lmsys.org/blog/2023-03-30-vicuna/.

[3] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168* (2021).

[4] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024).

[5] DeepSeek-AI. 2025. EPLB: Expert Parallelism Load Balancer. https://github.com/deepseek-ai/EPLB.

[6] DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. 2026. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. *arXiv preprint arXiv:2606.19348* (2026).

[7] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V. Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. 2022. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*.

[8] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[9] GLM-4.5 Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, Yean Cheng, Yifan An, Yilin Niu, Yuanhao Wen, Yushi Bai, Zhengxiao Du, Zihan Wang, Zilin Zhu, Bohan Zhang, Bosi Wen, Bowen Wu, Bowen Xu, Can Huang, Casey Zhao, Changpeng Cai, Chao Yu, Chen Li, Chendi Ge, Chenghua Huang, Chenhui Zhang, Chenxi Xu, Chenzheng Zhu, Chuang Li, Congfeng Yin, Daoyan Lin, Dayong Yang, Dazhi Jiang, Ding Ai, Erle Zhu, Fei Wang, Gengzheng Pan, Guo Wang, Hailong Sun, Haitao Li, Haiyang Li, Haiyi Hu, Hanyu Zhang, Hao Peng, Hao Tai, Haoke Zhang, Haoran Wang, Haoyu Yang, He Liu, He Zhao, Hongwei Liu, Hongxi Yan, Huan Liu, Huilong Chen, Ji Li, Jiajing Zhao, Jiamin Ren, Jian Jiao, Jiani Zhao, Jianyang Yan, Jiaqi Wang, Jiayi Gui, Jiayue Zhao, Jie Liu, Jijie Li, Jing Li, Jing Lu, Jingsen Wang, Jingwei Yuan, Jingxuan Li, Jingzhao Du, Jinhua Du, Jinxin Liu, Junkai Zhi, Junli Gao, Ke Wang, Lekang Yang, Liang Xu, Lin Fan, Lindong Wu, Lintao Ding, Lu Wang, Man Zhang, Minghao Li, Minghuan Xu, Mingming Zhao, Mingshu Zhai, Pengfan Du, Qian Dong, Shangde Lei, Shangqing Tu, Shangtong Yang, Shaoyou Lu, Shijie Li, Shuang Li, Shuang-Li, Shuxun Yang, Sibo Yi, Tianshu Yu, Wei Tian, Weihan Wang, Wenbo Yu, Weng Lam Tam, Wenjie Liang, Wentao Liu, Xiao Wang, Xiaohan Jia, Xiaotao Gu, Xiaoying Ling, Xin Wang, Xing Fan, Xingru Pan, Xinyuan Zhang, Xinze Zhang, Xiuqing Fu, Xunkai Zhang, Yabo Xu, Yandong Wu, Yida Lu, Yidong Wang, Yilin Zhou, Yiming Pan, Ying Zhang, Yingli Wang, Yingru Li, Yinpei Su, Yipeng Geng, Yitong Zhu, Yongkun Yang, Yuhang Li, Yuhao Wu, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yuxuan Zhang, Zezhen Liu, Zhen Yang, Zhengda Zhou, Zhongpei Qiao, Zhuoer Feng, Zhuorui Liu, Zichen Zhang, Zihan Wang, Zijun Yao, Zikang Wang, Ziqiang Liu, Ziwei Chai, Zixuan Li, Zuodong Zhao, Wenguang Chen, Jidong Zhai, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. 2025. GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models. *arXiv preprint arXiv:2508.06471* (2025).

[10] GLM-5-Team, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang,

Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui, Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. 2026. GLM-5: from Vibe Coding to Agentic Engineering. *arXiv preprint arXiv:2602.15763* (2026).

[11] Amos Goldman, Nimrod Boker, Maayan Sheraizin, Nimrod Admoni, Artem Polyakov, Subhadeep Bhattacharya, Fan Yu, Kai Sun, Georgios Theodorakis, Hsin-Chun Yin, Peter-Jan Gootzen, Aamir Shafi, Assaf Ravid, Salvatore Di Girolamo, Manjunath Gorentla Venkata, and Gil Bloch. 2026. NCCL EP: Towards a Unified Expert Parallel Communication API for NCCL. *arXiv preprint arXiv:2603.13606* (2026).

[12] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. 2022. FasterMoE: Modeling and Optimizing Training of Large-Scale Dynamic Pre-Trained Models. In *Proceedings of the 27th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[13] Pieter Hintjens. 2013. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of the Advances in Neural Information Processing Systems 32 (NeurIPS)*.

[15] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, HoYuen Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. 2023. Tutel: Adaptive Mixture-of-Experts at Scale. In *Proceedings of the 6th Conference on Machine Learning and Systems (MLSys)*.

[16] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. *arXiv preprint arXiv:2401.04088* (2024).

[17] Chao Jin, Ziheng Jiang, Zhihao Bai, Zheng Zhong, Juncai Liu, Xiang Li, Ningxin Zheng, Xi Wang, Cong Xie, Qi Huang, Wen Heng, Yiyuan Ma, Wenlei Bao, Size Zheng, Xuegui Zheng, Yanghua Peng, Haibin Lin, Xuanzhe Liu, Xin Jin, and Xin Liu. 2026. MegaScale-MoE: Large-Scale Communication-Efficient Training of Mixture-of-Experts Models in Production. In *Proceedings of the 21st European Conference on Computer Systems (EuroSys)*.

[18] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, M. C., Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y. Charles, H. S. Che, Guanduo Chen, Guangyu Chen, Guanzheng Chen, Huarong Chen, Jia Chen, Jianlong Chen, Jun Chen, Kexin Chen, Peng Chen, Ruijue Chen, Wentao Chen, Xin Chen, Yang Chen, Yanru Chen, Yifei Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Dazhi Cheng, Yean Cheng, Jialei Cui, Jingbing Cui, Anqi Dai, Jiaqi Deng, Hao Ding, Rui Ding, Shaofeng Ding, Mengfan Dong, Mengnan Dong, Yuhao Dong, Yuxin Dong, Angang Du, Chenzhuang Du, Dikang Du, Jusen Du, Yulun Du, Yu Fan, Jing Feng, Qiulin Feng, Yichen Feng, Kelin Fu, Qiang Fu, Fuxuan Gao, Hongcheng Gao, Jingyue Gao, Tong Gao, Weijia Gao, Shangyi Geng, Jie Gong, Linhu Gong, Shengao Gong, Xiaochen Gong, Qizheng Gu, Yicheng Gu, Shuhao Guan, Haiqing Guo, Shiqi Guo, Xiang Guo, Zhengyan Guo, Beixi Hao, Wenxin Hao, Xiaoru Hao, Dailan He, Haotian He, Lehan He, Qi He, Weiran He, Xinran He, Xinyi He, Yibo He, Yunjia He, Chao Hong, Tiange Hong, Hao Hu, Jiaxi Hu, Ruikun Hu, Weiming Hu, Yangyang Hu, Zhenxing Hu, Liang Hua, Jinbin Huang, Ke Huang, Ruiyuan Huang, Siying Huang, Weixiao Huang, Yan Huang, Zhengjie Huang, Zhiqi Huang, Yulong Hui, Chaobo Jia, Yutong Jiang, Zhejun Jiang, Zuoyou Jiang, Wenyi Jin, Xinyi Jin, Yu Jing, Huanjun Kong, Guokun Lai, Aidi Li, Cheng Li, Chengyuan Li, Cong Li, Fang Li, Guanyu Li, Haoyang Li, Jia Li, Junxiong Li, Lei Li, Letian Li, Lincan Li, Weihong Li, Wentao Li, Xintong Li, Yang Li, Yishen Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zhaoxi Li, Zheming Li, Zhengxiao Li, Zhiyuan Li, Jiawei Lin, Xiaohan Lin, Yibo Lin, Zichao Lin, Ziyan Lin, Bill Liu, Boxiao Liu, Chuan Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yipeng Liu, Zhengying Liu, Zhiheng Liu, Enzhe Lu, Haoyu Lu, Linqiang Lu, Tingzhan Lu, Zhiyuan Lu, Aotian Luo, G. Luo, Junyu Luo, Yifan Luo, B. Lyu, Wenzhou Lyu, Shaoguang Mao, Yuan Mei, Xin Men, Minqing Ni, Yixuan Niu, Siyuan Pan, Shujun Peng, Zhangyang Qi, Ruoyu Qin, ZeChao Qin, Zeyu Qin, Haiquan Qiu, Jianxin Qiu, Jiezhong Qiu, Bowen Qu, Yuhao Qu, Zeyu Shang, Youbo Shao, Han Shen, Jincheng Shi, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Wingchun Siu, Pengwei Song, Xiaoxi Song, Jianlin Su, Yunfeng Su, Zhaochen Su, Lin Sui, Jingsong Sun, Junyao Sun, Shaoning Sun, Shuzhe Sun, Tongyu Sun, Yujun Sun, Yunpeng Tai, Chuning Tang, Heyi Tang, Sirui Tang, Zecheng Tang, Chaoran Tian, Rongpeng Tian, Yu Tian, Wei Tu, Chensi Wang, Chuang Wang, Chunjie Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Hao Wang, Hao Wang, Huaqing Wang, Hui Wang, Jiayi Wang, Jinglong Wang, Jinhong Wang, Jiuzheng Wang, Linian Wang, Shaobo Wang, Shenzhi Wang, Shuyi Wang, Si Wang, Siyuan Wang, Tianfu Wang, Wenjue Wang, Xingran Wang, Xinmei Wang, Xinyuan Wang, Xusheng Wang, Yalin Wang, Yangkun Wang, Yao Wang, Yaoyu Wang, Yejie Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhenhao Wang, Zhongsheng Wang, Zifan Wang, Chu Wei, Ming Wei, Shouxin Wei, Zichen Wen, Fan Wu, Haoning Wu, Rucong Wu, Wenhao Wu, Xiaoxue Wu, Yingcong Wu, Yongqi Wu, Yuxin Wu, Zijian Wu, Xinglang Xian, Chenxuan Xiang, Yuye Xiang, Bocheng Xiao, Chenjun Xiao, Xin Xiao, Jin Xie, Xiaotong Xie, Yifeng Xie, Zhe Xie, Bowei Xing, Yiming Xiong, Baosheng Xu, Boyu Xu, Jiale Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Qingtao Xu, Shuyao Xu, Suting Xu, Tiantian Xu, Tianxiang Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ye Xu, Yueni Xu, Ziyao Xu, Haonan Xue, Junjie Yan, Yaoyao Yan, Fan Yang, Guangyao Yang, Hao Yang,

Junwei Yang, Ruoyu Yang, Wenjie Yang, Xiaofei Yang, Xinyu Yang, Yi Yang, Yiling Yang, Ying Yang, Yuchen Yang, Zhen Yang, Zhilin Yang, Zian Yang, Zuhao Yang, Haotian Yao, Dan Ye, Haoran Ye, Wenjie Ye, Zhanbo Ye, Bohong Yin, Haoxiang Yin, Xietong Yin, Chengzhen Yu, Haozhen Yu, Longhui Yu, Shengnan Yu, Shuying Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Tongtian Yue, Wei Yue, Yang Yue, Dunyuan Zha, Haobing Zhan, B. H. Zhang, Dehao Zhang, Fei Zhang, Hao Zhang, Haoyuan Zhang, Huanyu Zhang, Jiapei Zhang, Jiaxuan Zhang, Jin Zhang, Kaiyi Zhang, Miaozhen Zhang, Puqi Zhang, Qinglei Zhang, Rong Zhang, Rui Zhang, Shaoshuai Zhang, Shiyi Zhang, Xiaobin Zhang, Xiaoyun Zhang, Y. Zhang, Yangkun Zhang, Ye Zhang, Yichi Zhang, Yikun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Zijing Zhang, Bin Zhao, Chenguang Zhao, Feifan Zhao, Jinglun Zhao, Jinxiang Zhao, Shuai Zhao, Wenshuo Zhao, Xiangyu Zhao, Xuanle Zhao, Yikai Zhao, Zijia Zhao, Haozhi Zheng, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Haofeng Zhong, Lei Zhong, Longguang Zhong, M. Zhou, Qiankang Zhou, Runjie Zhou, Ruozhang Zhou, Xinyu Zhou, Yiqiao Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu, Yangjunfeng Zhu, Yuxuan Zhu, Zhen Zhu, Chen Zhuang, Weiyu Zhuang, and Xinxing Zu. 2026. Kimi K3: Open Frontier Intelligence. *arXiv preprint arXiv:2607.24653* (2026).

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

[20] Yiran Lei, Dongjoo Lee, Liangyu Zhao, Daniar Kurniawan, Chanmyeong Kim, Heetaek Jeong, Changsu Kim, Hyeonseong Choi, Liangcheng Yu, Arvind Krishnamurthy, Justine Sherry, and Eriko Nurvitadhi. 2026. FAST: An Efficient Scheduler for All-to-All GPU Communication. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[21] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *Proceedings of the 9th International Conference on Learning Representations (ICLR)*.

[22] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. 2023. Accelerating Distributed MoE Training and Inference with Lina. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC)*.

[23] Nandor Licker, Kevin Hu, Vladimir Zaytsev, and Lequn Chen. 2025. pplx-kernels: Perplexity MoE Kernels. https://github.com/perplexityai/pplx-kernels.

[24] Nandor Licker, Kevin Hu, Vladimir Zaytsev, and Lequn Chen. 2026. fabric-lib: RDMA Point-to-Point Communication for LLM Systems. In *Proceedings of the 9th Conference on Machine Learning and Systems (MLSys)*.

[25] Ziming Liu, Boyu Tian, Guoteng Wang, Zhen Jiang, Peng Sun, Zhenhua Han, Tian Tang, Xiaohe Hu, Yanmin Jia, Yan Zhang, He Liu, Mingjun Zhang, Yiqi Zhang, Qiaoling Chen, Shenggan Cheng, Mingyu Gao, Yang You, and Siyuan Feng. 2026. Expert-as-a-Service: Towards Efficient, Scalable, and Robust Large-scale MoE Serving. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. To appear.

[26] Ziming Mao, Yihan Zhang, Chihan Cui, Zhen Huang, Kaichao You, Zhongjie Chen, Zhiying Xu, Zhenyu Gu, Scott Shenker, Costin Raiciu, Yang Zhou, and Ion Stoica. 2026. UEP: Portable Expert-Parallel Communication. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[27] Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Pete Walsh, Oyvind Tafjord, Nathan Lambert, Yuling Gu, Shane Arora, Akshita Bhagia, Dustin Schwenk, David Wadden, Alexander Wettig, Binyuan Hui, Tim Dettmers, Douwe Kiela, Ali Farhadi, Noah A. Smith, Pang Wei Koh, Amanpreet Singh, and Hannaneh Hajishirzi. 2025. OLMoE: Open Mixture-of-Experts Language Models. In *Proceedings of the 13th International Conference on Learning Representations (ICLR)*.

[28] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[29] NVIDIA Corporation. 2024. A low-latency GPU memory copy library based on NVIDIA GPUDirect RDMA technology. https://github.com/NVIDIA/gdrcopy.

[30] OpenAI. 2025. gpt-oss-120b & gpt-oss-20b Model Card. *arXiv preprint arXiv:2508.10925* (2025).

[31] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 51st International Symposium on Computer Architecture (ISCA)*.

[32] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST)*.

[33] Qwen Team. 2024. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115* (2024).

[34] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*.

[35] Daniele De Sensi, Tiziano De Matteis, Konstantin Taranov, Salvatore Di Girolamo, Tobias Rahn, and Torsten Hoefler. 2022. Noise in the Clouds: Influence of Network Performance Variability on Application Scalability. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 3 (2022), 49:1–49:27.

[36] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*.

[37] Rong Shi, Sreeram Potluri, Khaled Hamidouche, Jonathan Perkins, Mingzhe Li, Davide Rossetti, and Dhabaleswar K. Panda. 2014. Designing efficient small message transfer mechanism for inter-node MPI communication on InfiniBand GPU clusters. In *Proceedings of the 21st International Conference on High Performance Computing (HiPC)*.

[38] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[39] Mohammad Siavashi, Faezeh Keshmiri Dindarloo, Dejan Kostic, and Marco Chiesa. 2025. Priority-Aware Preemptive Scheduling for Mixed-Priority Workloads in MoE Inference. In *Proceedings of the 5th Workshop on Machine Learning and Systems (EuroMLSys)*.

[40] StepFun, Bin Wang, Bojun Wang, Changyi Wan, Guanzhe Huang, Hanpeng Hu, Haonan Jia, Hao Nie, Mingliang Li, Nuo Chen, Siyu Chen, Song Yuan, Wuxun Xie, Xiaoniu Song, Xing Chen, Xingping Yang, Xuelin Zhang, Yanbo Yu, Yaoyu Wang, Yibo Zhu, Yimin Jiang, Yu Zhou, Yuanwei Lu, Houyi Li, Jingcheng Hu, Ka Man Lo, Ailin Huang, Binxing Jiao, Bo Li, Boyu Chen, Changxin Miao, Chang Lou, Chen Hu, Chen Xu, Chenfeng Yu, Chengyuan Yao, Daokuan Lv, Dapeng Shi, Deshan Sun, Ding Huang, Dingyuan Hu, Dongqing Pang, Enle Liu, Fajie Zhang, Fanqi Wan, Gulin Yan, Han Zhang, Han Zhou, Hanghao Wu, Hangyu Guo, Hanqi Chen, Hanshan Zhang, Hao Wu, Haocheng

Zhang, Haolong Yan, Haoran Lv, Haoran Wei, Hebin Zhou, Heng Wang, Heng Wang, Hongxin Li, Hongyu Zhou, Hongyuan Wang, Huiyong Guo, Jia Wang, Jiahao Gong, Jialing Xie, Jian Zhou, Jianjian Sun, Jiaoren Wu, Jiaran Zhang, Jiayu Liu, Jie Cheng, Jie Luo, Jie Yan, Jie Yang, Jieyi Hou, Jinguang Zhang, Jinlan Cao, Jisheng Yin, Junfeng Liu, Junhao Huang, Junzhe Lin, Kaijun Tan, Kaixiang Li, Kang An, Kangheng Lin, Kenkun Liu, Lei Yang, Liang Zhao, Liangyu Chen, Lieyu Shi, Liguo Tan, Lin Lin, Lin Zhang, Lina Chen, Liwen Huang, Liying Shi, Longlong Gu, Mei Chen, Mengqiang Ren, Ming Li, Mingzhe Chen, Na Wang, Nan Wu, Qi Han, Qian Zhao, Qiang Zhang, Qianni Liu, Qiaohui Chen, Qiling Wu, Qinglin He, Qinyuan Tan, Qiufeng Wang, Qiuping Wu, Qiuyan Liang, Quan Sun, Rui Li, Ruihang Miao, Ruosi Wan, Ruyan Guo, Shangwu Zhong, Shaoliang Pang, Shengjie Fan, Shijie Shang, Shilei Jiang, Shiliang Yang, Shiming Hao, Shuli Gao, Siming Huang, Siqi Liu, Tiancheng Cao, Tianhao Cheng, Tianhao Peng, Wang You, Wei Ji, Wen Sun, Wenjin Deng, Wenqing He, Wenzhen Zheng, Xi Chen, Xiangwen Kong, Xianzhen Luo, Xiaobo Yang, Xiaojia Liu, Xiaoxiao Ren, Xin Han, Xin Li, Xin Wu, Xu Zhao, Yanan Wei, Yang Li, Yangguang Li, Yangshijie Xu, Yanming Xu, Yaqiang Shi, Yeqing Shen, Yi Yang, Yifei Yang, Yifeng Gong, Yihan Chen, Yijing Yang, Yinmin Zhang, Yizhuang Zhou, Yuanhao Ding, Yuantao Fan, Yuanzhen Yang, Yuchu Luo, Yue Peng, Yufan Lu, Yuhang Deng, Yuhe Yin, Yujie Liu, Yukun Chen, Yuling Zhao, Yun Mou, Yunlong Li, Yunzhou Ju, Yusheng Li, Yuxiang Yang, Yuxiang Zhang, Yuyang Chen, Zejia Weng, Zhe Xie, Zheng Ge, Zheng Gong, Zhenyi Lu, Zhewei Huang, Zhichao Chang, Zhiguo Huang, Zhirui Wang, Zidong Yang, Zili Wang, Ziqi Wang, Zixin Zhang, Binxing Jiao, Daxin Jiang, Heung-Yeung Shum, and Xiangyu Zhang. 2025. Step-3 is Large yet Affordable: Model-system Co-design for Cost-effective Decoding. *arXiv preprint arXiv:2507.19427* (2025).

[41] Benjamin Thérien, Charles Étienne Joseph, Zain Sarwar, Ashwinee Panda, Anirban Das, Shi-Xiong Zhang, Stephen Rawls, Sambit Sahu, Eugene Belilovsky, and Irina Rish. 2025. Continual Pre-training of MoEs: How robust is your router? *Transactions on Machine Learning Research* (2025).

[42] Shaoyu Wang, Guangrong He, Geon-Woo Kim, Yanqi Zhou, and Seo Jin Park. 2025. Toward Cost-Efficient Serving of Mixture-of-Experts with Asynchrony. *arXiv preprint arXiv:2505.08944* (2025).

[43] Shaoyu Wang, Yizhuo Liang, Jaeyong Song, Chong Li, and Seo Jin Park. 2026. Moebius: Serving Mixture-of-Expert Models with Seamless Runtime Parallelism Switch. *arXiv preprint arXiv:2606.26607* (2026).

[44] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[45] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388* (2025).

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

[47] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, Quan Chen, and Xin Liu. 2025. Comet: Fine-grained Computation-communication Overlapping for Mixture-of-Experts. In *Proceedings of the 8th Conference on Machine Learning and Systems (MLSys)*.

[48] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. 2025. DeepEP: An Efficient Expert-Parallel Communication Library. https://github.com/deepseek-ai/DeepEP.

[49] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Proceedings of the Advances in Neural Information Processing Systems 37 (NeurIPS)*.

[50] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[51] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. 2025. MegaScale-Infer: Efficient Mixture-of-Experts Model Serving with Disaggregated Expert Parallelism. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*.

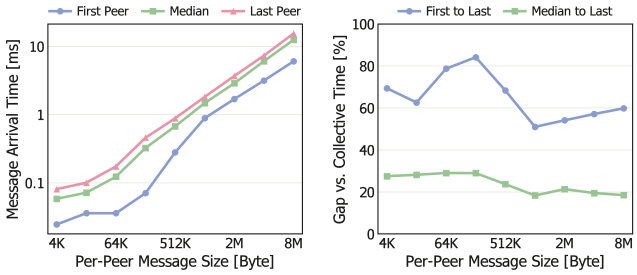

Figure 18. Per-peer arrival times on an 8-node, 16-GPU cluster. Left: first, mean, and last peer arrival. Right: the first-to-last gap and the median-to-last gap as fractions of the collective's duration.

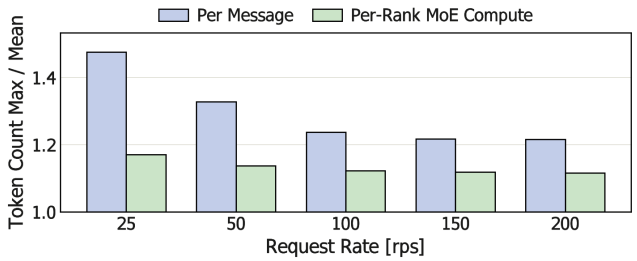

Figure 19. Token-count imbalance of communication and MoE compute within each decoding step, averaged over all steps and layers; 1.0 is perfect balance.

Appendices

Appendices are supporting materials that have not been peer-reviewed.

A Varying Message Arrival Times

Figure 2 reports the message sizes that a decoding workload generates in our cluster. Figure 18 extends the same instrumented all-to-all to more message sizes.

B Message-Level Load Imbalance

§A fixes every peer's message size to isolate the network's contribution to straggling. Figure 19 measures the imbalance of the traffic itself, that is, how many tokens need to be transferred from one rank to another during the collectives between DP-attention layers and EP-MoE layers. We compare the maximum and the mean number of tokens in such messages of every decode step. Notably, with EPLB-like expert placement, the max-to-mean gap of each rank's MoE compute load is minimal, while the gap of tokens to send is more significant.

For this experiment, due to testbed availability issues, we use a scaled-down version (4 nodes, 8 GPUs) of the aforementioned Sphere cluster to run the SGLang baseline with half of the layers of GPT-OSS-120B. The token routing pattern is preserved by methods described in §D.

C Detailed Model Configuration

Table 6 shows the detailed configurations of the models we tested in the evaluation.

Table 6. Model configurations.

	GPT-OSS-120B	GLM-4.5-Air
Total / Active params	117B / 5.1B	106B / 12B
Experts	128	128 + 1 shared
Top-k	4	8
Layers	36	46
Hidden dim	2,880	4,096
Expert intermediate dim	2,880	1,408
Head dim	64	128
Attn heads / KV heads	64 / 8	96 / 8

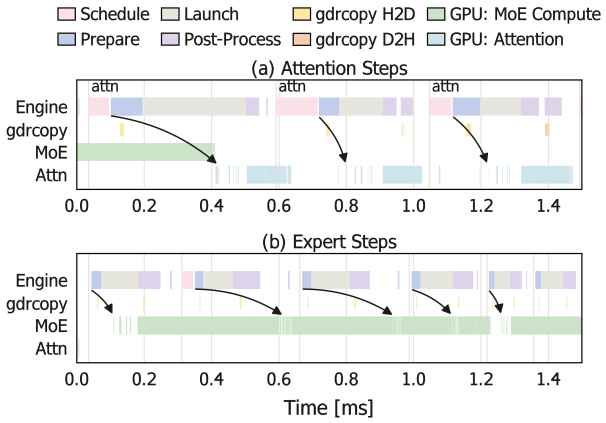

Figure 20. Captured StreamInfer trace examples for GPT-OSS-120B running on the Sphere testbed in bf16. Arrows point from a CPU scheduling operation to the corresponding GPU execution.

D Model Serving Details

Dummy KV-Cache. StreamInfer is a decoding-only system, so every request's prompt KV has to come from somewhere: either a prefill instance streams it in or a KV-Cache storage service sends it in. Both options are orthogonal to the barrier-free decoding technique. Since KV transfer techniques differ wildly in performance and can impact the decoding performance, we serve with dummy cache tensors for both StreamInfer and the baselines to isolate the decoding component alone. We preserve the model arithmetic, the data movement, and the sequence length distribution. This setup also lets us cut the layer count when the testbed size is limited.

MoE Routing Profiles. Dummy cache and cutting layers cannot produce meaningful gating decisions, so we profile the per-sequence, per-token routing outcomes of serving the datasets using the actual model and replay them by request and token position during the experiments. In this way, a half-layer configuration also inherits the same routing behavior as the full model.

EPLB-Like Expert Placement. Relying on the routing profile replays, we further balance each dataset's profile by rearranging which experts live on which rank, so that MoE compute is close to perfectly balanced across ranks, and both StreamInfer and the baselines serve using this placement.

EP Collectives. The version of SGLang used for evaluation does not support all-to-all EP execution on the evaluation hardware. It rather employs all-gather/all-reduce for running MoE models like the GPT-OSS series.

E Example Traces

Figure 20 shows the two exemplary segments from the trace used to present Table 4. It shows that the attention computes are shorter and more sparse, while expert computes are longer and bulkier because they are dominated by the up and down projections. This explains why the overheads for scheduling expert batches are more adequately hidden behind computes. It also shows that, when an attention batch follows an expert batch, more of the overheads are successfully overlapped.