xDeepServe: Model-as-a-Service on Huawei CloudMatrix384

XDS Team @ Huawei

The rise of scaled-out LLMs and scaled-up SuperPods signals a new era in large-scale AI infrastructure. LLMs continue to scale out via MoE, as seen in recent models like DeepSeek, Kimi, and Qwen. In parallel, AI hardware is scaling up, with Huawei's CloudMatrix384 SuperPod offering hundreds of GB/s high-speed interconnects. Running large MoE models on SuperPod-scale hardware brings new challenges. It requires new execution models, scalable scheduling, efficient expert load balancing, and elimination of single points of failure.

This paper presents xDeepServe, Huawei Cloud's LLM serving system designed for SuperPod-scale infrastructure. At its core is Transformerless, a disaggregated architecture that decomposes transformer models into modular units—attention, feedforward, and MoE—executed independently on NPUs connected via high-speed fabric. We implement this design in two forms: disaggregated prefill-decode and disaggregated MoE-attention. This fully disaggregated setup enables independent scaling of compute and memory without sacrificing performance. To support this architecture, we propose XCCL, a communication library that leverages CloudMatrix384's global shared memory to implement efficient point-to-point and all-to-all primitives. We also extend our serving engine FlowServe with system-level techniques, enabling scalable inference across hundreds of NPUs.

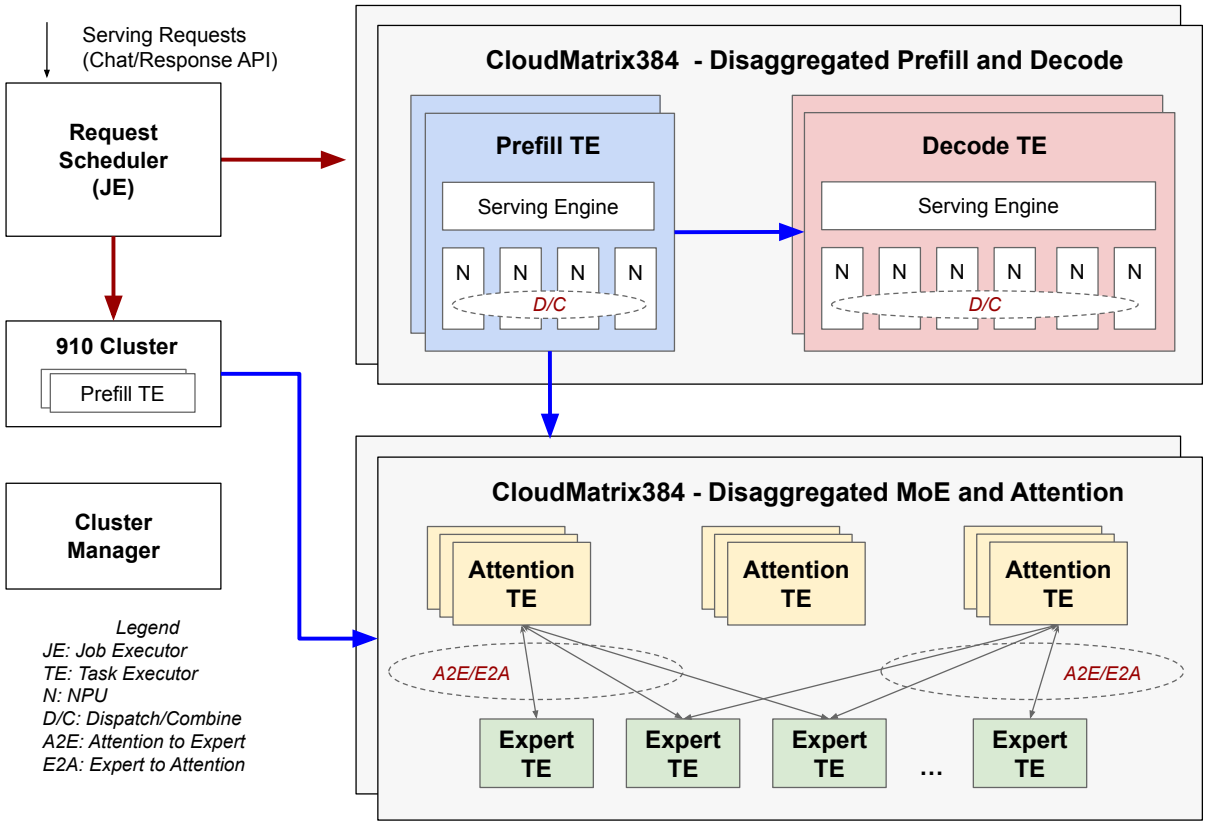


Figure 1 | **xDeepServe over CloudMatrix384 SuperPod.** We deploy both disaggregated Prefill–Decode and disaggregated MoE–Attention across heterogeneous clusters. Red lines indicate requests sent to prefill, blue lines represent requests forwarded from prefill to decode.

Contents

1 Introduction **3**

2 Background **4**
2.1 xDeepServe 4
2.2 CloudMatrix384 5
2.3 NPU Execution Mode 6

3 Communication Library over CloudMatrix's Distributed Shared Memory **6**
3.1 Point-to-Point Primitives 7
3.2 All-to-All Communication for Colocated MoE-Attention 8
3.3 All-to-All Communication for Disaggregated MoE-Attention 10

4 Scalable Serving System at SuperPod-Scale **12**
4.1 Overview 12
4.2 Scalable Data Parallel Group 13
4.3 DP Load Balancing 14
4.4 Proactive Garbage Collection to Reduce Jitter 15
4.5 Expert Load Balancing 15
4.6 Multi-Token Prediction 18
4.7 INT8 Quantization of DeepSeek Models 18

5 Transformerless: Towards Fully Disaggregated LLM Serving **20**
5.1 Disaggregated Prefill and Decode 21
5.2 Disaggregated MoE and Attention 22
5.3 Vision: Dataflow Serving at SuperPod-Scale 24

6 Reliability **24**
6.1 Failure Detection 24
6.2 Failure Recovery 25

7 Evaluation **26**
7.1 Decode Performance 26
7.2 Production Workload 27

8 Conclusion **28**

9 Contributors **28**

1. Introduction

Large language models (LLMs) keep growing in size and complexity. To improve quality and efficiency, many recent models, such as DeepSeek [14], Kimi K2 [11], and Qwen [22], adopt the Mixture-of-Experts (MoE) architecture. MoE scales out model capacity by activating only a small subset of experts per token, reducing compute cost while keeping high model quality. At the same time, AI hardware is scaling up. Huawei's CloudMatrix384 connects 384 Ascend chips with a hundreds of GB/s high-speed interconnect, offering a tightly coupled memory and compute environment across the entire pod.

When these two trends meet—scaled-out MoE models and scaled-up SuperPods—new challenges emerge. MoE models require fine-grained expert routing, synchronization, and load balancing across hundreds of NPU devices. SuperPod hardware enables global shared memory and uniform low-latency access across hundreds of NPUs, but traditional LLM serving systems are not built to exploit these properties. To unlock the full potential of SuperPod-scale infrastructure, we need a new system that runs disaggregated workloads, maintains low tail latency, and scales efficiently across hundreds of NPUs.

This paper presents **xDeepServe**, Huawei Cloud's in-house LLM serving system for SuperPod-scale deployment. We advocate resource disaggregation [18] as a foundational design principle for large-scale serving. Disaggregation decouples system components into independently scalable units, improves fault isolation, and enables flexible system evolution. To realize these benefits, we introduce *Transformerless*, an architecture that breaks transformer-based LLMs into modular components—attention, feedforward, and MoE—and runs each module independently on dedicated NPUs. This design decouples compute and memory, reduces interference across stages, and supports separate scaling.

We integrate two forms of disaggregation.

First, to reduce interference between compute-bound prefill and memory-bound decode phases, we use disaggregated prefill-decode architecture (§5.1). While prior systems like Splitwise [16], TetriServe [7], and DistServe [24] explored this direction, our design targets the unique demands of large MoE models with expert parallelism on SuperPod-scale hardware. This disaggregated pipeline is further enhanced by heterogeneous deployment: we run prefill on both Ascend 910 (previous generation) and CloudMatrix384 NPUs, while decode runs exclusively on CloudMatrix's nodes to leverage high-bandwidth interconnects. This design balances cost and performance, while achieving high throughput and low latency at scale.

Second, we further disaggregate MoE and attention, running them on separate NPUs to reduce resource contention and improve utilization (§5.2). While prior work explored this idea on smaller models, xDeepServe is the first to apply it at SuperPod scale, deploying DeepSeek-V3/R1 across 768 Ascend NPU dies, with 288 handling MoE and 480 handling attention. This separation introduces a compute imbalance: MoE is stateless and scales with batch size, while attention maintains KV cache and scales with sequence length. To address this, we introduce three techniques: (1) a trampoline-based `A2E/E2A` routing protocol to handle asymmetric NPU allocation, (2) a new *DP domain* abstraction to enable inter-group scheduling without shrinking batch sizes, and (3) persistent kernel scheduling across concurrent compute and communication streams to eliminate CPU overhead. These designs allow us to fully utilize both MoE and attention NPUs while maintaining sub-200 μs dispatch latency across the entire SuperPod.

To support fully disaggregated execution, we propose XCCL, a novel communication library that provides memory-semantic point-to-point and all-to-all primitives over CloudMatrix384's shared memory fabric. XCCL supports microsecond-level latency and high concurrency across more than 300,000 NPU pairs. We use it to build KV cache transfer, MoE dispatch/combine, and MoE-attention (§3).

xDeepServe further builds a scalable serving engine at SuperPod-Scale called FlowServe. It introduces the *Data Parallel (DP) group* abstraction, where each group manages its own execution pipeline—including tokenization, SPMD execution, caching, and networking—without relying on central coordination. FlowServe scales to hundreds of NPUs by eliminating single points of scalability bottlenecks and applying system-level techniques to reduce latency under MoE barriers such as expert `dispatch` and `combine`. These include load-aware request routing among DPs, proactive garbage collection to reduce jitter, and MoE expert load balancing. To improve efficiency and hardware utilization, FlowServe supports Multi-Token Prediction (MTP) and INT8 quantization, enabling high-throughput inference under tight SLA constraints.

To ensure reliability at SuperPod scale, xDeepServe incorporates reliability mechanisms across failure detection and recovery paths. It uses a multi-level heartbeat and link-probing system to detect faults ranging from hung processes to silent KV-transfer stalls. For recovery, xDeepServe evolves from coarse-grained cluster restarts to fine-grained component-level resilience. It supports independent failover of prefill and decode stages, dynamic reconfiguration of expert ranks, and selective token recomputation on transient network or memory faults. These designs allow the system to maintain availability and throughput even under hardware disruptions, a critical capability for large-scale MoE serving.

xDeepServe is deployed in production to serve large-scale DeepSeek [14], Kimi [11], and Qwen [22] models. It achieves 2400 tokens per second per Ascend NPU chip while meeting a 50s time-per-output-token (TPOT) SLA. In the rest of the paper, we first describe the architecture of CloudMatrix384 in §2.2, followed by the low-level communication library in §3. We then present the serving engine in §4 and the Transformerless serving architecture in §5. Finally, we discuss our reliability mechanisms in §6.

2. Background

2.1. xDeepServe

xDeepServe is Huawei Cloud's fully-hosted, serverless LLM serving platform, designed to support large-scale, multi-tenant workloads. It evolves from the DeepServe system introduced in our previous work [9], with new capabilities targeting LLM serving at the scale of CloudMatrix384. xDeepServe has been in production for over a year, operating on a large Ascend NPU cluster. It provides industry-standard APIs for fine-tuning, agent hosting, and model serving. Motivated by the growing demand for generative AI services (e.g., ChatGPT and DeepSeek), xDeepServe addresses key production challenges, including heterogeneous workload durations, distributed stateful execution, and dynamic resource demands.

To tackle these challenges, xDeepServe introduces four core components [9]. First, it adopts a serverless abstraction based on a request–job–task model to manage diverse AI workloads, spanning post-training and inference tasks. Second, it integrates a custom serving engine, FlowServe,

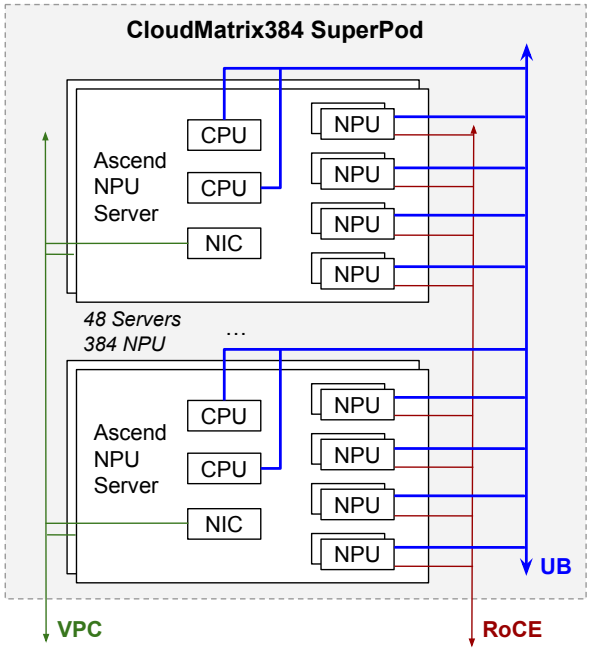


Figure 2 | **An Overview of CloudMatrix384 SuperPod.** *A CloudMatrix384 SuperPod has 48 servers and 384 Ascend NPU chips in total. A single CloudMatrix384 Ascend NPU chip has two dies interconnected via high-bandwidth NoC. The chip uses the decoupled DaVinci architecture.*

which follows a microkernel-inspired architecture, supports NPU-centric execution, and uses SPMD-style parallelism for high-performance LLM serving. Third, it includes scheduling policies designed to support both prefill-decode (PD) disaggregated and PD-colocated deployment modes. Fourth, it incorporates system-level optimizations, such as pre-warmed pods, DRAM preloading, and NPU fork, to enable rapid elasticity, scaling to 64 instances within seconds.

While our previous paper [9] introduced the core architecture of xDeepServe, this report focuses on the new challenges and techniques arising from deployment on the CloudMatrix384 SuperPod. We present our latest system designs and methods for leveraging its scaled-up capabilities to enable next-generation LLM serving.

2.2. CloudMatrix384

Figure 2 shows an overview of CloudMatrix384. A CloudMatrix384 SuperPod is a single scale-up domain composed of 48 servers, delivering hundreds of PFLOPs of FP16 compute, several terabytes of on-chip memory, and terabytes per second of memory bandwidth. Each server in the SuperPod is equipped with multiple CPUs, multiple NICs, and 8 Ascend NPU chips. Crucially, the SuperPod integrates three types of network fabrics. The first is a traditional VPC network that connects servers to external systems or cloud services. The second is a scaled-out RoCE network that interconnects all NPU chips and can be extended across multiple SuperPods and previous generation Ascend NPU servers. The third is a scaled-up UB network that makes CloudMatrix384 a SuperPod. It connects all CPUs and NPUs within the SuperPod in an all-to-all fashion, offering roughly several times the bandwidth of RoCE. See [10, 13, 26] for more.

The scaled-up UB network enables several novel properties. First, it provides a global shared

memory address space spanning all CPU DRAM and NPU on-chip memory, allowing any chip to access the memory of any other chip. This access supports two semantics: traditional DMA and memory semantics, which will be detailed later. Second, the system eliminates intra-server NUMA locality—CPUs have uniform access latency to all eight NPUs within a single server.

2.3. NPU Execution Mode

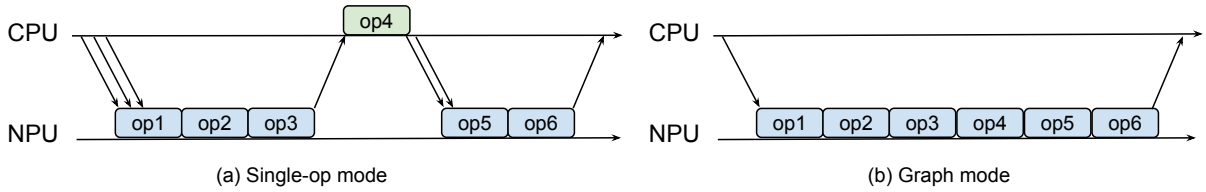


Figure 3 | **Compare Single-Op and Graph Mode Execution on Ascend NPU.**

We describe the two execution modes supported by Ascend NPUs and how we apply them at different stages of LLM serving to balance flexibility and performance. Frameworks, such as PyTorch, represent neural networks as computation graphs, where operators are nodes and data dependencies are edges. These graphs can run in two modes: *single-op* and *graph*.

Single-op Mode. This is PyTorch's default execution mode. Each operator is dispatched to the NPU operator queue upon Python invocation, and its result is accessible on next synchronization. This dynamic style supports arbitrary control flow, dynamic tensor shapes, and native debugging, and it allows limited CPU–NPU concurrency when dependencies permit. However, it introduces overhead because each operator launch incurs host-device communication, and the NPU may stay idle if the operator execution time is lower than the dispatch time of next operator. That said, for computation-heavy operators, this cost is acceptable. Therefore, we use this mode during prefill to handle dynamic input shapes.

Graph Mode. Graph mode traces and compiles a static computation graph, which is then dispatched to the NPU in a single kernel launch. This eliminates per-operator launch overhead and improves device utilization. Huawei's TorchAir (based on PTA) enables graph-mode inference on Ascend NPUs. As shown in Figure 3(b), the CPU submits the entire computation graph in one operation, reducing dispatch overhead. We use this mode during decode, where inputs are small and mostly static, to achieve optimal performance.

3. Communication Library over CloudMatrix's Distributed Shared Memory

A high-performance communication library is essential to fully harness the scaled-up UB fabric in CloudMatrix384. We introduce XCCL, a communication library purpose-built for LLM serving on SuperPod. XCCL proposes distributed memory protocols over CloudMatrix384's global shared memory to construct efficient networking primitives. Its protocol design resembles classical far-memory systems based on one-siden RDMA verbs [3, 20]. Note that the memory transfer units (mtu) in NPU's computing cores, along with the DMA engines, are fundamental to enabling global shared memory in CloudMatrix384. Both can read from or write to the on-chip memory of any other NPU in the SuperPod via the UB fabric. This capability underpins our customized communication library for serving. In general, memory transfer units are used for

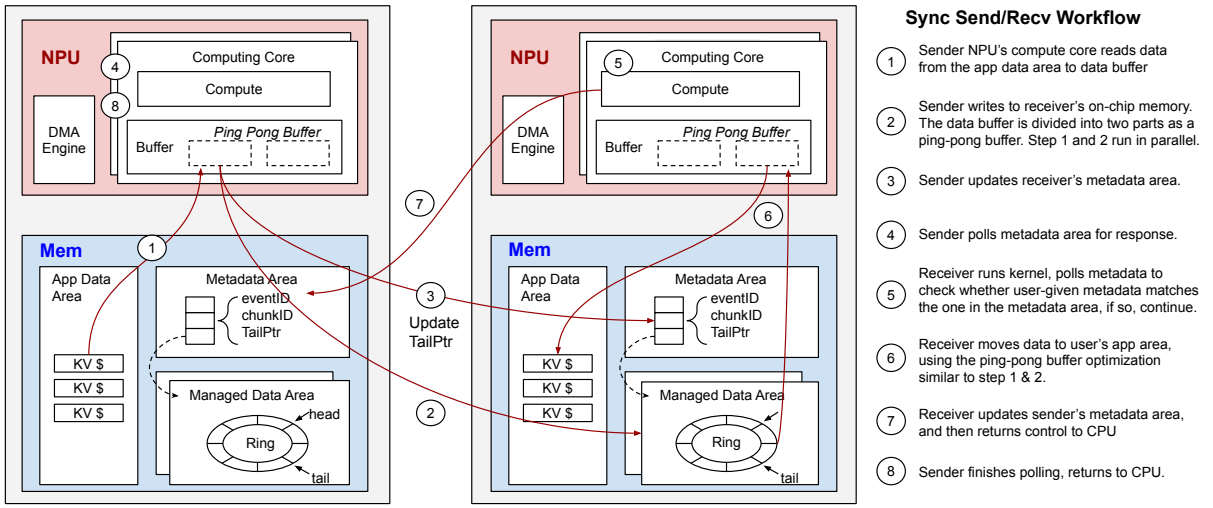


Figure 4 | **Distributed Send/Recv Workflow.** *We only show memory-semantic-based transfer using memory transfer units while memory copy between on-chip memory can also be done by using the DMA engine. We also have a zero-copy version in which the send and recv kernel directly manipulate the app data area.*

low-latency communication, while DMA engines are preferred for high-throughput transfers. This is because memory transfer units provide a memory-semantic API and move data between the computing core's data buffer and on-chip memory, with transfers limited to the buffer size of hundreds of KBs. In contrast, DMA engines use DMA semantics and support bulk data movement up to several GBs.

Below, we describe the point-to-point APIs used in disaggregated prefill and decode (§3.1), the dispatch and combine APIs for expert parallelism (EP) (§3.2), and the A2E/E2A APIs for disaggregated MoE and attention (§3.3).

3.1. Point-to-Point Primitives

This section describes the design and implementation of point-to-point APIs such as `send` and `receive` on the CloudMatrix384 SuperPod. These APIs are essential for serving, enabling efficient data transfer in scenarios such as disaggregated prefill and decode, sequence parallelism, and fast scaling via `npu-fork` [9]. For simplicity, we use the transfer of KV cache [8] from prefill NPU to decode NPU to illustrate the `send`/`receive` design. XCCL's `send` and `receive` operations can occur between any pair of NPUs in CloudMatrix384, hence our design can scale to roughly 300K potential pairs.

Data structure. Each NPU's on-chip memory is partitioned into three areas: the app data area, the metadata area, and the managed data area. The **app data area** is reserved for application-specific data such as KV cache, used directly by our serving engine. The **metadata area** stores control information required for sanity checks and to implement the distributed memory protocol. It consists of metadata fields, one for each pair of computing cores to enable parallelism among computing cores. Given that CloudMatrix384 includes 384 NPU chips, each with 2 dies and up to 48 computing cores per die, the total number of fields is $384 \times 2 \times 48 \times 2 \approx 74$K. Each 32-byte field includes a user-defined `eventID` for sanity checking, a kernel-generated `chunkID`

for tracking chunked transfers, and a `tailPtr` pointing to the ring buffer in the data area. The total metadata size is set to 4 MB. The **managed data area** is used for data exchange between NPUs. A dedicated ring buffer is maintained for each NPU pair, with a fixed number of slots of fixed size per buffer. Both the metadata and managed data areas are managed by XCCL. The current implementation is not zero-copy; data is copied between the app data area and the managed data area when XCCL APIs are invoked.

Distributed Memory Protocol. We design an distributed protocol to transfer data from sender to receiver over CloudMatrix384's global shared memory, illustrated in Figure 4. **Step 1:** The sender's serving engine invokes XCCL's `send`, passing the source buffer in the app data area (e.g., KV cache), an `eventID` (e.g., number of sends), the receiver NPU's ID, and the number of computing cores to use. XCCL launches a kernel on the sender NPU. The send kernel uses mtu-in to copy data from the app data area to each computing core's data buffer in parallel. **Step 2:** The send kernel then reads the destination's on-chip memory address from its metadata area and uses mtu-out to transfer data from the data buffer to the receiver's managed data area. Each data buffer operates in a ping-pong fashion, allowing mtu-in and mtu-out to run concurrently. Alternatively, the send kernel can also directly copy data from the app data area to the receiver's managed data area directly using the DMA engine, which bypass the data buffer but at the cost of higher starting latency. **Step 3:** The send kernel updates the receiver's `tailPtr` in the metadata area via mtu-out, indicating the amount of data transferred. **Step 4:** The send kernel then busy-polls its local metadata area for an acknowledgment from the receiver. **Step 5:** Meanwhile, the receiver's serving engine invokes XCCL's `receive`, passing the destination buffer in the app data area, the `eventID`, the sender NPU's ID, and the number of computing cores. XCCL launches a kernel on the receiver NPU, which polls the metadata area for new data. **Step 6:** Upon detecting data, the receive kernel copies it from the managed data area to the app data area using mtu-in and mtu-out, again in a ping-pong fashion. **Step 7:** After the transfer completes, the receive kernel updates the sender's metadata area to acknowledge completion, then returns control to the CPU and the `receive` caller. **Step 8:** The send kernel detects the acknowledgment, returns control to the CPU, and completes the `send` call. In addition to this synchronous protocol, XCCL supports an asynchronous mode in which send and receive kernels avoid busy polling.

Performance. Our `send` and `recv` primitives achieve microsecond-level latency. Figure 5 evaluates their performance under varying data sizes and numbers of computing cores per kernel invocation. We measure end-to-end latency, from the start to the completion of `send`, including all protocol steps described earlier. We randomly select two NPU dies at different SuperPod servers, leveraging the uniform bandwidth of the scaled-up UB fabric. As shown in Figure 5, for payloads smaller than 1 MB, latency remains under 20 μs even with just 2 computing cores. As the data size increases, performance scales with parallelism: transferring 9 MB using all 48 computing cores is more than 2.5$\times$ faster than using only 2.

3.2. All-to-All Communication for Colocated MoE-Attention

This section presents the design and implementation of two all-to-all communication primitives—`dispatch` and `combine`—used in large-scale expert parallelism (EP). In EP, `dispatch` routes each token's hidden state to its top-k experts based on gating scores, while `combine` aggregates expert outputs, weighted by the same scores, into a unified tensor. Together, we

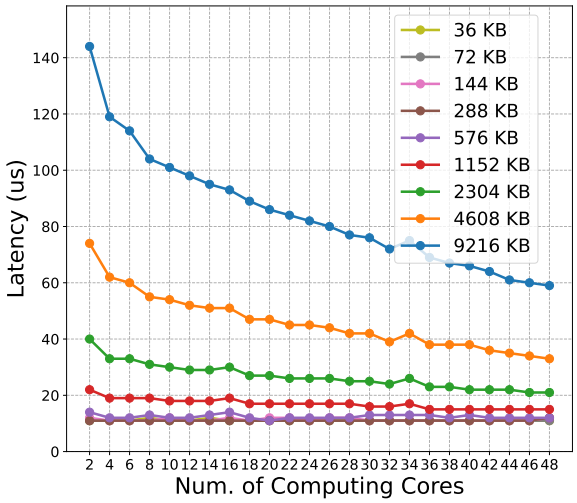


Figure 5 | **Evaluation of Send/Recv.** *We vary the data size and the number of computing cores used for a single send/recv pair.*

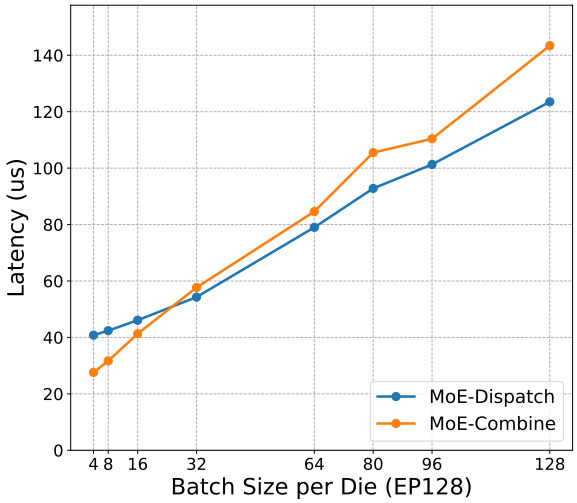


Figure 6 | **Evaluation of Dispatch/Combine.** *We vary the batch size per die with a fixed EP128. We use DeepSeek-R1.*

find these primitives account for at least 25% of MoE execution time. Similar to point-to-point primitives, we implement these all-to-all operations using far memory semantics rather than traditional network-level verbs like DeepEP [2]. For brevity, we detail the `dispatch` protocol below; `combine` follows a similar pattern.

Data Structure. Similar to the point-to-point primitives, we partition each NPU's on-chip memory into three regions. The metadata area contains 32-byte fields, one per rank. Each field includes an event ID for sanity checking, a pointer to the corresponding rank's buffer offset in the managed data area, and a token count received from that rank. The max number of metadata fields is similar to that of point-to-point primitives. The managed data area is partitioned by rank ID, with each NPU assigned a fixed-size block determined by the maximum batch size.

Distributed Memory Protocol of `dispatch`. The `dispatch` protocol consists of two main phases: broadcasting the number of tokens each rank should receive, followed by the actual data transfer. Figure 7 illustrates the complete workflow. **Step 1:** The sender's serving engine invokes `dispatch`, passing the source buffer in the app data area, an event ID, and additional parameters. A kernel is launched on the sender NPU, which uses mtu-in to copy data from the app data area to each computing core's data buffer in parallel. **Step 2:** If quantization is enabled, the kernel converts the data from FP16/BF16 to INT8 using certain instructions. **Step 3:** The kernel writes token data into the managed data area, partitioned by destination rank ID. **Step 4:** The kernel updates each destination rank's metadata with the number of tokens it will receive. **Step 5:** Each NPU polls its local metadata area until metadata from all ranks is received. **Steps 6–7:** Once all metadata is received, the kernel pulls token data from peer NPUs into the data buffer using mtu-in, guided by received offsets and token counts. It then copies the data into the destination buffer in the app data area via mtu-out.

Performance. Figure 6 evaluates `dispatch` and `combine`'s performance under varying batch sizes per die, using DeepSeek-R1 models with a fixed EP128 configuration. At small batch sizes, `dispatch` exhibits slightly higher latency than `combine` due to the additional overhead

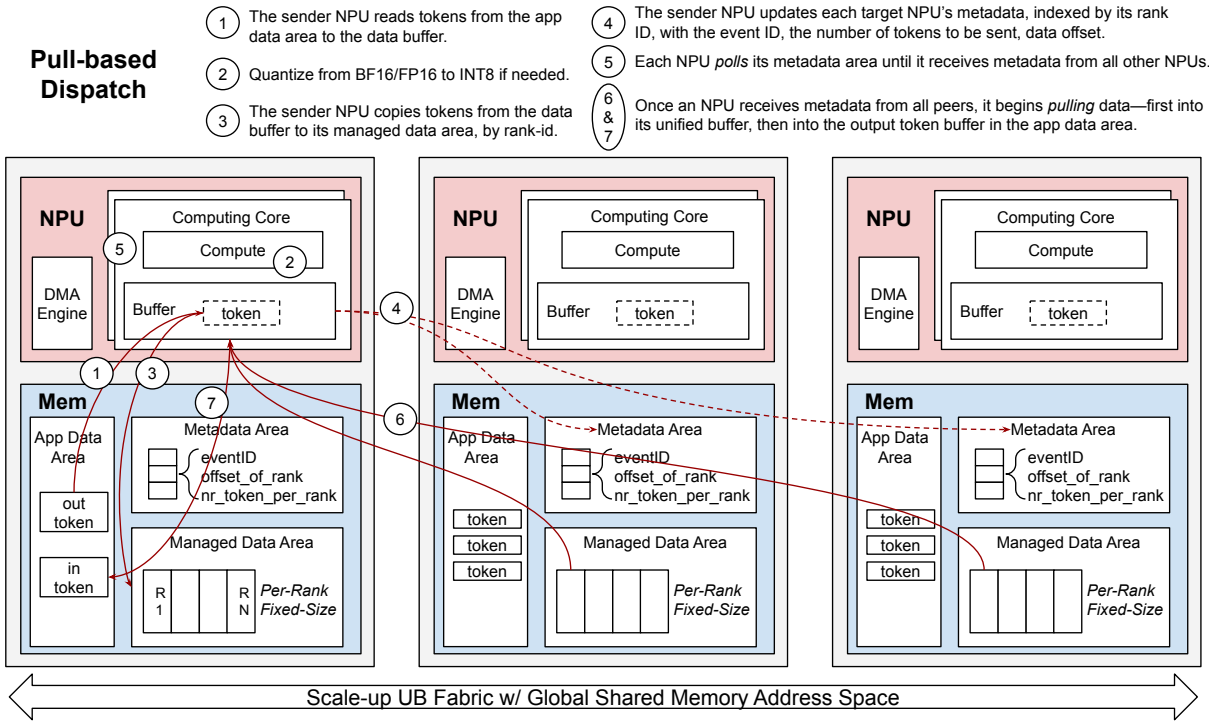


Figure 7 | **Pull-based Dispatch based on UB's Global Shared Memory.** *Dashed red lines are metadata transfer. Bulk data transfer in step 6 and 7 can also use the DMA engine. A single dispatch kernel can use multiple computing cores. All NPUs involved in a dispatch run the same seven steps described above. We show three NPUs for simplicity.*

from quantization. However, since quantization reduces data size by half, `dispatch` becomes faster than `combine` when the batch size per die exceeds 32. For reference, with a batch size per die of 96 and EP128, the total global batch size across the system reaches $96 \times 128 = 12{,}288$.

3.3. All-to-All Communication for Disaggregated MoE-Attention

This section presents the design of two all-to-all communication primitives—`attention2expert` (`A2E`) and `expert2attention` (`E2A`)—used in disaggregated MoE and attention deployments (see §5.2). These primitives enable efficient token routing and aggregation between attention NPUs and expert NPUs at scale.

Similar to `dispatch`, the `A2E` primitive routes each token's hidden state from the attention NPUs to its top-k expert NPUs, as determined by the gating scores. Conversely, `E2A` functions like `combine`, collecting outputs from the selected experts and merging them—weighted by the same gating scores—back at the attention NPUs. Both primitives use the same distributed shared memory infrastructure and share similar data structures with `dispatch`/`combine`.

A key difference, however, is that attention and expert modules reside on separate NPU dies, introducing asymmetry in resource allocation. In large-scale deployments, this asymmetry presents new challenges. For example, in a typical DeepSeek-R1/V3 [14] configuration with 288 experts, we often provision 288 expert NPUs but only 160 attention NPUs (see §5.2). In a naive pull-based `dispatch` design, each attention NPU would push metadata for all expert NPUs and

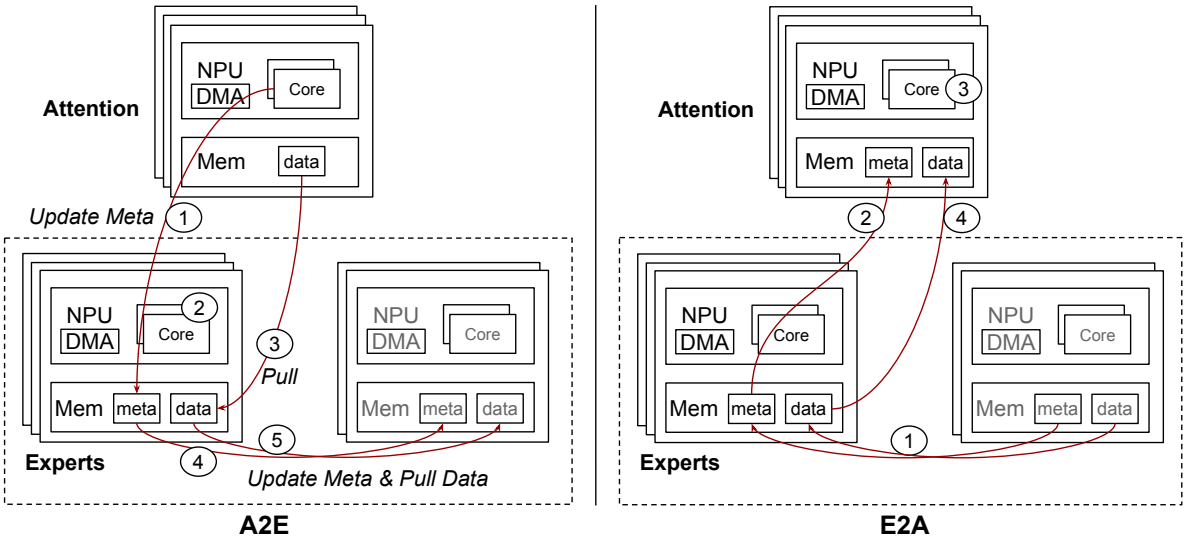


Figure 8 | **Attention 2 Expert and Expert 2 Attention Primitives.** *In A2E, the leftmost set of expert NPUs serves as trampolines, receiving data directly from the attention NPUs. In E2A, expert outputs are first routed to the trampoline NPUs, which then forward the data to the attention NPUs. This two-stage routing reduces metadata updates and balances communication across asymmetric NPU allocations.*

wait for them to pull data. This approach quickly becomes inefficient due to the high fan-out and limited scalar throughput of each computing core.

Trampoline Forward. To address this, we introduce a novel design called trampoline forward. In this scheme, a subset of expert NPUs—equal in number to the attention NPUs—is designated as trampolines. These trampoline NPUs first receive all data from the attention NPUs and then forward it to the remaining expert NPUs. This two-stage routing reduces metadata overhead and balances traffic across the asymmetrically allocated NPUs. The complete data flow is illustrated in Figure 8.

Trade-off between Memory Semantic and DMA. To improve communication efficiency, we employ `NPU-Direct Unified Remote Memory Access (URMA)`, a technique on Ascend NPUs similar to IBGDA on GPUs [14]. `NPU-Direct URMA` enables computing cores to issue remote memory access requests directly to the DMA engine, bypassing both the host CPU and AI CPU as shown in Figure 2.2. Although `NPU-Direct URMA` incurs higher startup latency compared to memoty transfer units, it offers three key advantages. First, it reduces computing core consumption. By offloading memory access to the DMA engine using asynchronous semantics, more computing core resources become available for communication or computation. Second, it is better suited for high-throughput scenarios. Memory transfer units transfers are limited by the computing core's data buffer size, whereas the DMA engine supports transfers up to several GBs. Third, it avoids contention with compute streams. Since mtu-in is also used for computation, using DMA helps prevent interference when compute and communication streams share the same NPU die, as discussed in §5.2.

Performance. We evaluate `A2E` and `E2A` at SuperPod scale using a single deployment with three DP domains, each containing 160 DP groups (TP = 1), and 288 expert NPUs. We run DeepSeek-R1 with a per-die batch size of 96, resulting in a global batch size of $96 \times 3 \times 160 =$

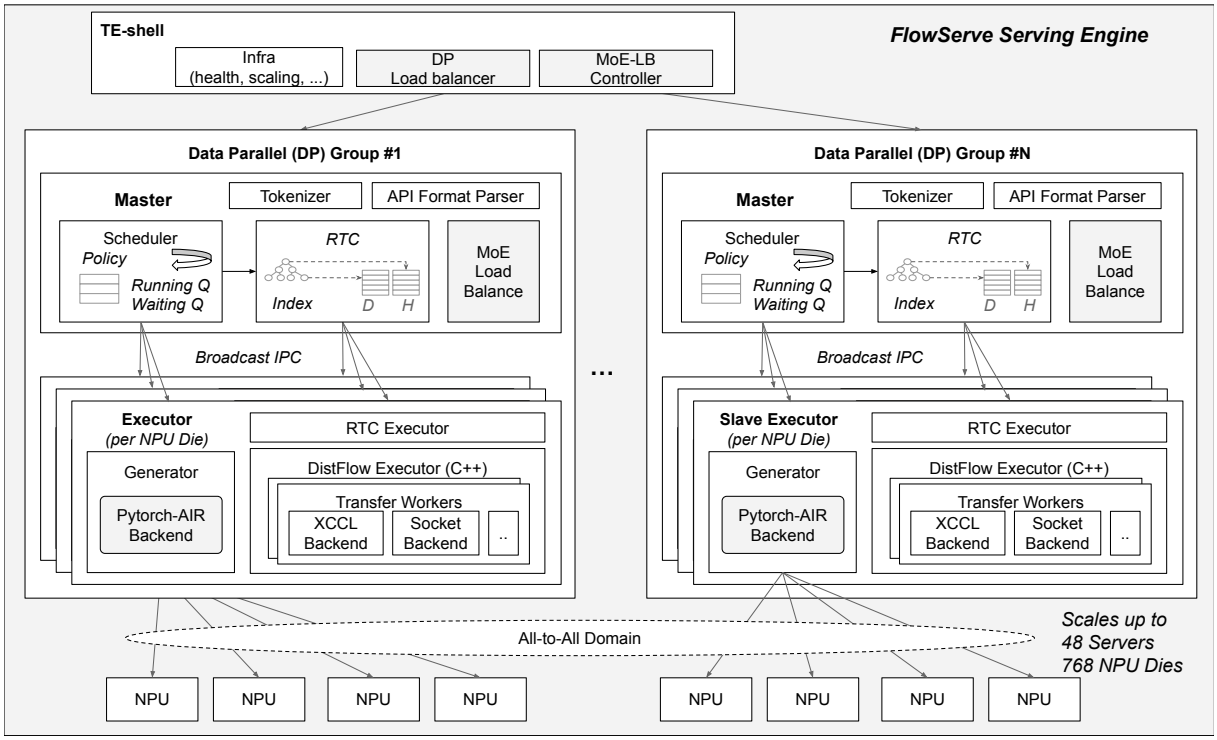


Figure 9 | **The Architecture of FlowServe.** *A single FlowServe engine can span an entire CloudMatrix384 SuperPod—48 Ascend NPU servers with 768 NPU dies. FlowServe scales at the granularity of a DP group, where each group encapsulates a complete serving pipeline. To avoid bottlenecks and single points of failure, both request scheduling and response handling are fully distributed across DP groups.*

46,080. Under this configuration, A2E achieves a latency of 172 μs, while E2A completes in 193 μs. These results demonstrate that both primitives maintain low-latency execution even under high concurrency and large-scale deployment.

4. Scalable Serving System at SuperPod-Scale

4.1. Overview

The LLM model scales out via MoE; the hardware scales up via the CloudMatrix384 SuperPod. At Huawei Cloud, our goal is to scale the serving system to efficiently run large-scale MoE models on SuperPod-scale infrastructure. Scaling introduces new challenges: it requires the right abstractions, efficient scheduling, and the elimination of both performance bottlenecks and single points of failure.

To this end, we redesign our serving engine, FlowServe [9], evolving it from a single-node deployment to a SuperPod-scale distributed system, as illustrated in Figure 9 and Figure 10. This redesign centers on three key components:

- **First**, we introduce the *Data Parallel (DP) group* abstraction, inspired by SGLang [23]. Each DP group encapsulates a full serving pipeline—including tokenization, API parsing, SPMD-style executors, the Relational Tensor Cache (RTC), and the DistFlow networking

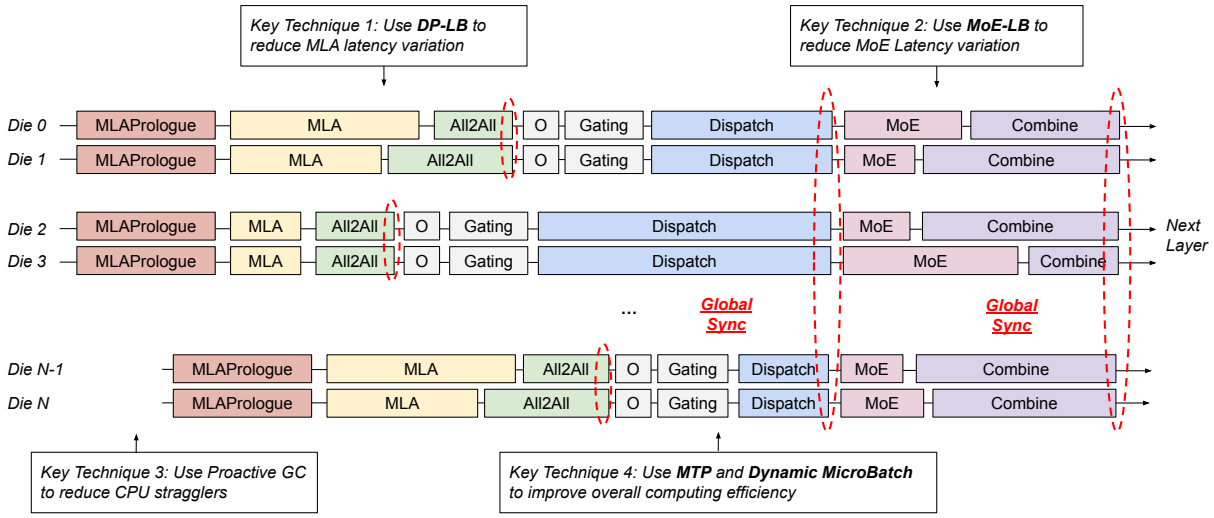


Figure 10 | **The Execution Timeline of Running DeepSeek @ FlowServe.** *This figure illustrates a single MoE layer, highlighting four key challenges it introduces along with our solutions. An all-to-all follows the MLA attention because we sometimes run MLAPrologue and MLA with TP=1 to avoid KV cache duplication, while executing the output projection with TP> 1 to accelerate computation.*

stack [9]. To eliminate bottlenecks and single points of failure, both request scheduling and response handling are fully decentralized across DP groups.

- **Second**, we implement system-level optimizations to minimize latency from global synchronization. Although requests are scheduled independently across DP groups spanning hundreds of NPUs, MoE models introduce two global barriers: `dispatch` and `combine`, which can severely limit scalability. To address this, we apply DP-level load balancing and proactive garbage collection to reduce `dispatch` latency variance, and MoE load balancing to optimize `combine`.
- **Third**, we optimize model execution to improve forward-pass efficiency. We implement efficient Multi-Token Prediction (MTP) and Dynamic MicroBatching to better utilize hardware and increase throughput when serving DeepSeek-R1/V3.

We describe these optimizations below.

4.2. Scalable Data Parallel Group

During the decode phase of MoE model serving, FlowServe adopts an expert parallel (EP) pattern for MoE layers and a data parallel (DP) pattern for attention layers. As the number of DPs can scale to hundreds, a single FlowServe instance is capable of generating hundreds of thousands of tokens per second. The key challenge is to evolve FlowServe into a decentralized architecture without any single point of scalability bottleneck.

Figure 9 illustrates the evolved architecture of FlowServe. The core design principle for scalability is to make each DP a self-contained software stack and to eliminate cross-DP communication. Key components—including the scheduler, output processing module, RTC engine, and EP load-balancing (EP-LB) module (§4.5)—are replicated within each DP. The TE-shell acts as a centralized orchestrator for cross-DP coordination, but its responsibilities are limited to three

essential functions: dispatching requests across DPs (§4.3), triggering expert load balancing (§4.5), and coordinating health checks among all DPs (§6.1).

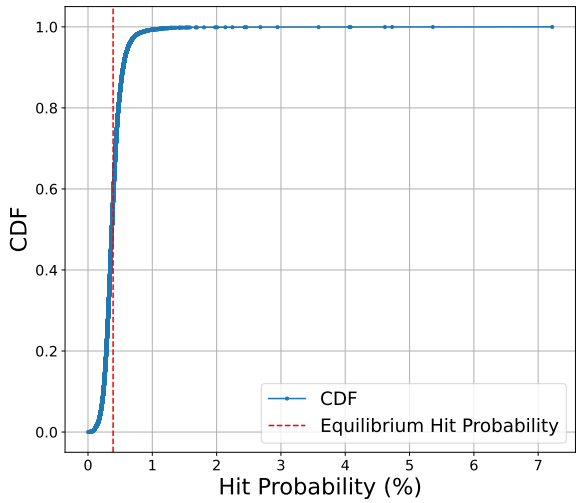
In FlowServe, each request is dispatched via the DP load balancer, centralized in the TE-shell, to achieve optimized load balancing using a global view. This dispatch occurs only once per request, making it affordable for the TE-shell. However, for streaming outputs, decentralization is essential. To address this, FlowServe introduces an *output shortcutting* optimization: the master process of each DP spawns a separate child process dedicated to output handling—including detokenization and output stream parsing (e.g., extracting reasoning content or tool-call results)—and relays the resultant messages directly to the xDeepServe frontend.

4.3. DP Load Balancing

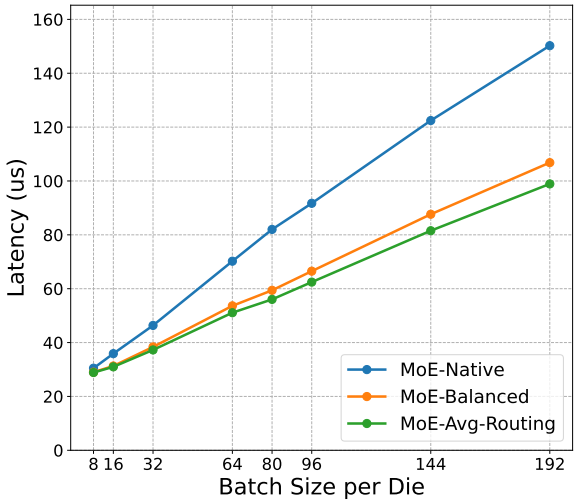
In large-scale model inference systems with multiple DPs, user requests are dispatched across several attention data parallel groups that share a common MoE backbone. The dispatch step introduces an implicit synchronization barrier: each DP group must wait at every MoE dispatch point for the slowest group to complete its previous attention block computations. If attention computation times are not evenly distributed, faster DP groups will idle, waiting for stragglers, which increases overall tail latency at the MoE dispatch point. Therefore, carefully balancing request routing across DP groups is crucial to prevent these MoE synchronization delays and to maintain high throughput.

Prefill DP Load Balancing. For the prefill phase, we adopt a novel single-level scheduling strategy. Our initial design extended the legacy single-DP engine to a two-level scheduler, where each request was first routed to a DP queue, and each DP ran its own local scheduler. However, this often led to stragglers—for example, one DP might pick a short batch while another handles a long one—resulting in poor utilization. Chunk-prefill mitigates this but adds chunking overhead and requires adaptive sizing. Sequence-parallel (SP) execution posed another challenge: long requests triggered full-DP participation, but reusing the same SP ranks for both 32K and 128K sequences proved inefficient. To balance prefill load, FlowServe adopts a single-level, collaborative scheduler. All tokenized requests are shared across DPs, and a leader scheduler (at `DP-0`) collects DP status via `all-gather` at each step. It then assigns request batches using a cost model (e.g., prefix cache hit rate). This global view ensures coordinated decisions across DPs. Unlike decoupled TE-shell schedulers, this approach provides timely and accurate scheduling, invoked only when pending requests exist.

Decode DP Load Balancing. In the decode phase, each DP group's load is shaped by two factors: the number of concurrent requests it handles and the memory consumed by the KV cache. Each DP group supports a fixed batch size. When full, it must block incoming requests until others finish, which increases both time-to-second-token (TTST) and TPOT. At the same time, KV cache usage reflects the memory footprint of active sequences. Uneven usage across groups leads to unbalancd MLA execution time as shown in Figure 10 or even trigger swapping, severely degrading performance. To balance decode load, we first exclude DP groups that have reached their batch limit. Among the rest, we select the group with the lowest KV cache usage, accounting for reserved space needed for long outputs. The TE-shell tracks real-time metrics for each group: it updates the pending request count on dispatch and completion, and collects periodic KV cache stats. This design enables informed, system-wide scheduling decisions without introducing significant overhead.



(a) *We show the expert load distribution of a DeepSeek-R1 layer under the ShareGPT workload. The distribution is highly skewed—20% of experts receive more than the average load, and the hottest expert sees 30× more tokens than the average.*



(b) *The setup uses EP288 and 1K sequence. MoE-Avg-Routing, which forces uniform load across all experts; MoE-Native, which uses the original token-to-expert assignment; and MoE-Balanced, which applies our EPLB to balance expert load.*

Figure 11 | **A Study of Expert Placement Load Balancing**.

4.4. Proactive Garbage Collection to Reduce Jitter

At SuperPod scale, we observe significant graph launch jitter as FlowServe's deployment scale grows. This jitter is most pronounced at the first `dispatch` operator (e.g., the fourth layer in DeepSeek models or the second in Kimi K2), where global synchronization across all dies occurs for the first time. In some cases, this jitter can exceed 100 ms. The issue is further aggravated by large-scale expert parallelism, where a single forward pass may involve hundreds of NPU and CPU cores—making the system highly sensitive to stragglers.

To mitigate this jitter, we employ three key optimizations:

- **Core pinning:** Each executor is pinned to a dedicated CPU core, minimizing kernel scheduling noise and context-switch overhead.
- **PTA caching:** PyTorch Air (PTA) is configured to cache compiled graphs, bypassing expensive runtime guard checks and reducing launch latency.
- **Manual Python GC:** Python's garbage collector is invoked manually at controlled intervals (e.g., after every few hundred forward passes) to prevent unpredictable pauses during critical dispatch operations.

4.5. Expert Load Balancing

Expert load balancing is critical for large-scale Expert Parallelism (EP) serving on CloudMatrix384. For DeepSeek models, we deploy EP288, consisting of 256 routed experts and 32 shared experts. Load imbalance among experts leads to performance degradation, causing slowdowns across all 288 NPUs due to straggler effects. Figure 11 illustrates this issue.

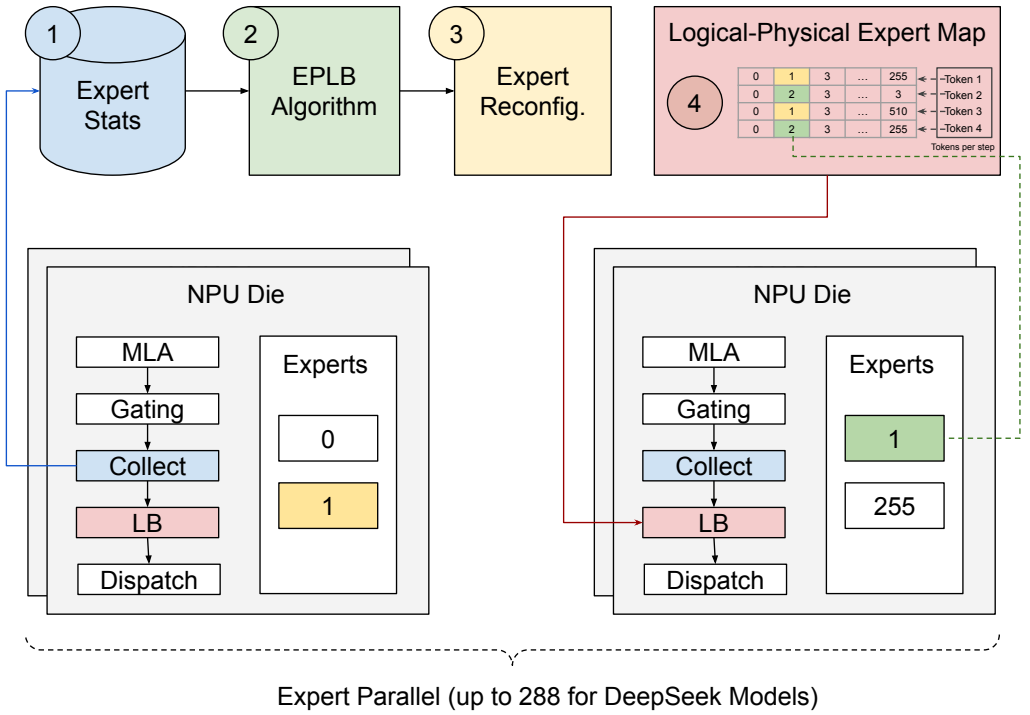


Figure 12 | **An Overview of FlowServe's Expert Load Balancing.** *It consists of four main components. First, expert load collection gathers token statistics across NPUs. Second, the EPLB algorithm selects redundant experts and determines their placement. Third, the system updates the expert-to-NPU mapping to reflect new replicas. Finally, during the forward pass, tokens are evenly distributed across expert replicas based on the updated mapping.*

To address expert load imbalance, we use a data-driven approach. Our design periodically analyzes token routing patterns, identifies overloaded ("hot") experts, and replicates these experts across multiple NPUs. Each MoE layer reserves redundant slots per NPU to host replicated experts. During inference, we evenly distribute tokens to these replicas using precomputed mappings. Meanwhile, we continuously collect and analyze activation data to update expert replica assignments. Expert weights are swapped asynchronously to maintain uninterrupted inference throughput under significant load imbalance.

Our design includes five key components, as shown in Figure 12.

Step 1: Collecting Expert Load Distribution. First, we collect data on expert loads across NPUs. We define expert load as the total number of tokens routed to each expert within a given time interval. Token count directly reflects both communication overhead (MoE-Dispatch and MoE-Combine) and computation workload (Expert MatMul). In each MoE layer, we insert a special `Collect` kernel after gating to track the number of tokens assigned to each expert per NPU. These counts are copied to on-chip memory's buffer. The executor of each NPU gathers these counts within its DP group and sends the aggregated data periodically (e.g., every minute) to FlowServe's TE shell. Frequent data collection incurs overhead, so we limit its frequency to balance accuracy and efficiency. Once we have gathered this expert load data, we proceed to determine which experts need replication.

Step 2: EPLB Algorithm & Assignment. Given the expert load data, we use the Expert

Placement Load Balancing (EPLB) algorithm to select redundant experts per layer to balance workloads. We define the hottest expert in each time slice t for layer ℓ as:

$$h_{\ell,t} = \arg\max_{e} \text{ token\_count}[\ell][e][t].$$

Then, the total load for layer ℓ is:

$$L_\ell = \sum_{t \in T} \text{token\_count}[\ell][h_{\ell,t}][t].$$

Given a redundancy budget R, we select redundant experts as follows:

1. Compute the current total load L_ℓ.
2. For each candidate expert c identified as hot in any time slice, simulate splitting tokens evenly across its replicas and compute the resulting total load $L_\ell(c)$.
3. Select the candidate expert c^* that minimizes the simulated load, and add it to the redundancy list.
4. Update token counts for the selected expert to reflect even distribution among its replicas.

With redundant experts selected, we next assign them efficiently to NPUs. To determine expert placement, we first calculate each redundant expert's total load by summing token counts across all time slices. We then sort these experts by load, starting from the highest. We assign each expert in order to the least-loaded NPU with available redundancy slots, updating that NPU's load after each assignment.

Step 3: Redundant Expert Reconfig. After assigning redundant experts to NPUs, we dynamically update their configurations in four phases. First, we prefetch new expert weights from storage into memory. Second, we temporarily disable redundant expert slots by modifying the logical-to-physical expert mapping. Third, we asynchronously load the prefetched weights into the target redundant slots. Finally, we restore the logical-to-physical mapping, re-enabling the redundant expert slots. This approach ensures seamless weight updates without interrupting ongoing inference tasks. Once reconfiguration is complete, subsequent forward passes use the updated mapping to evenly distribute tokens across expert replicas.

Step 4: Balancing Token Loads Across Expert Replicas In the final step, we ensure that tokens are evenly distributed across expert replicas during model forward. This requires mapping logical expert IDs—selected by the gating mechanism—to physical expert IDs representing actual NPU locations, while keeping both latency and communication overhead low. Directly collecting expert activation counts from all NPUs is too expensive. To address this, we adopt two practical solutions: **(1) Efficient Mapping with `gather` Operation.** We use PyTorch's `gather` operator to efficiently map logical experts to physical replicas. This operator translates the gating matrix into actual expert assignments in parallel, minimizing computational overhead. **(2) Communication-Free Load Balancing.** We implement decentralized load balancing by rotating token assignments across replicas based on each token's position in the batch. This ensures even distribution without requiring inter-NPU communication. As Figure 11b shows, we improve forward latency more than 40%.

Figure 12 illustrates how logical experts are mapped to rotated physical replicas based on token positions. Consider a case with 4 tokens per inference step and 256 logical experts. The

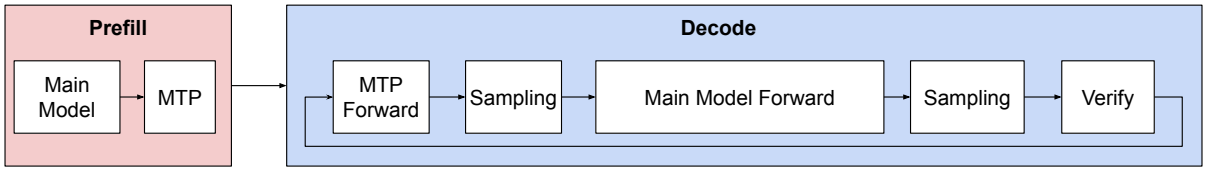


Figure 13 | **An Overview of FlowServe's MTP Execution Workflow.**

mapping table has shape *(4, 256)*, where each row corresponds to a token and each column to a logical expert. Suppose Logical Expert 1 has two replicas: a primary in slot 2 and a redundant in slot 1. The first column of the mapping table rotates between slots 1 and 2 across the 4 tokens, assigning each with equal probability to ensure balanced routing.

4.6. Multi-Token Prediction

This section describes how FlowServe integrates MTP into its disaggregated prefill-decode workflow (§5) to improve decode efficiency. Figure 13 illustrates the full execution pipeline. DeepSeek's Multi-Token Prediction (MTP) [14] extends conventional next-token prediction by training the model to generate multiple future tokens in sequence. During inference, MTP enables speculative decoding by predicting several tokens in one pass and verifying them afterward, accelerating autoregressive generation.

Execution Workflow. In the prefill stage, both the main model and MTP process the input prompt to construct the key-value (KV) cache and generate the first token. The KV cache and relevant hidden states are then transferred to the decode stage. During decoding, we run a tightly optimized five-step loop designed to eliminate CPU-induced latency bubbles. Our initial implementation followed the EAGLE framework from vLLM [12], but its default scheduling introduced noticeable stalls. We replaced it with a customized pipeline that maximizes NPU utilization and minimizes idle time. The loop proceeds as follows: (1) Run MTP forward to generate k draft tokens; (2) Sample token candidates from the MTP outputs; (3) Verify the draft tokens using the main model; (4) Sample again from the main model outputs; (5) Check final logits to decide token acceptance.

Multiple MTPs. DeepSeek publicly released parameters for only a single MTP layer. In practice, this layer achieves a 70%–90% acceptance rate across most workloads, reducing latency by up to 40% at fixed batch size. To support deeper speculation, we initially reused the released weights for a second MTP layer without retraining. This naive setup yielded just 2.26 tokens per step on production datasets. To improve performance, we trained a dedicated second MTP while freezing both the main model and the original MTP. Using FlowServe, we generated 280,000 samples from internal prompts. After training, the second MTP achieved 2.35 tokens per step—a 9% improvement over the reused baseline.

4.7. INT8 Quantization of DeepSeek Models

The Ascend NPU in CloudMatrix384 does not natively support FP8 arithmetic. To deploy DeepSeek-R1/V3—originally trained in FP8—we quantize the model to INT8 using Post-Training Quantization (PTQ) method. Our method integrates SmoothQuant [21] and GPTQ [4] techniques specifically for DeepSeek models.

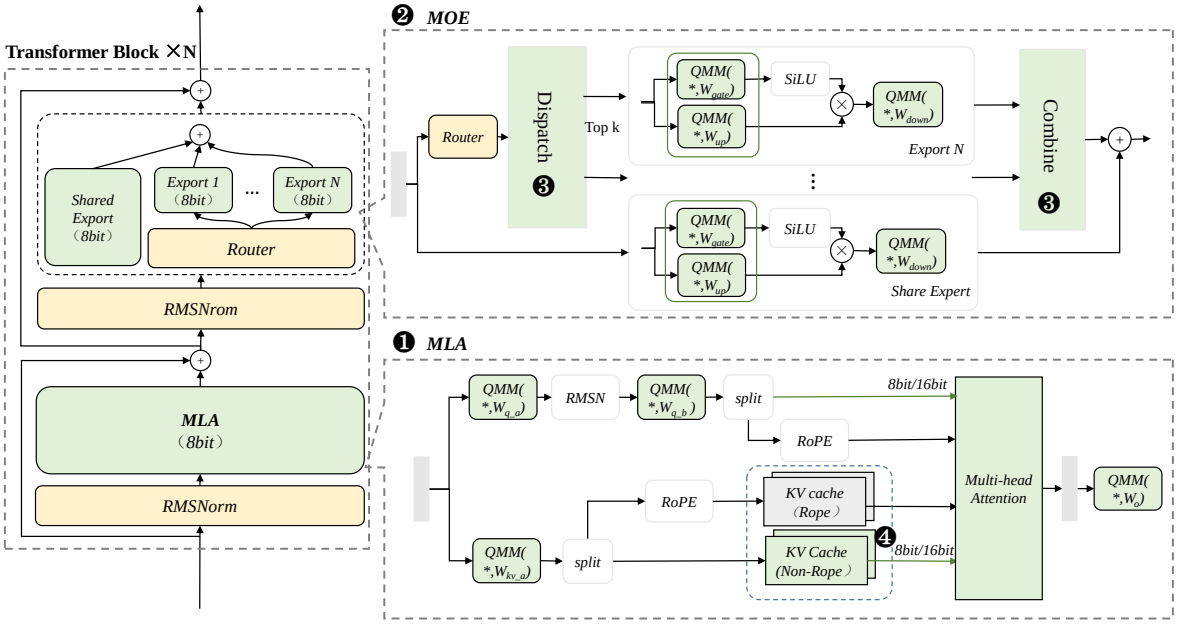


Figure 14 | **An Overview of INT8 Quantization in DeepSeek Models.** *We use optimized quantization strategies for MLA and MLP/MoE components, and fused quantization in MoE-dispatch communication to minimize overhead.*

We apply INT8 quantization to the model's MLA, MoE, and MLP modules. To minimize accuracy loss, we introduce error compensation during quantization. Activations use token-wise quantization (one scale per token), and weights use channel-wise quantization (one scale per output channel). At inference time, we leverage the hardware-accelerated `npu_quant_matmul` (QMM) operator for efficient INT8 matrix multiplication.

Figure 14 summarizes our quantization approach, described in four key components:

MLA Quantization. The MLA module uses low-rank compression for the `Q`, `K`, and `V` matrices and integrates RoPE. We quantize important weights to INT8, including query compression (`Wq_a`), key/value compression (`Wkv_a`), query reconstruction (`Wq_b`), and attention output projection (`Wo`). Our analysis shows activations have a significantly wider dynamic range (10-100×) compared to weights. To handle this, we apply a smoothing operation that redistributes quantization difficulty between activations and weights before quantization. Additionally, we use a GPTQ-based channel-wise quantization method with Hessian-guided iterative refinement. This approach dynamically updates the remaining FP weights to compensate for quantization errors. Figure 15 shows the input activation and weight magnitude in a DeepSeek-R1 linear layer, before and after smoothing to reduce quantization outliers.

MLP/MoE Quantization. DeepSeek models use MLP layers in early stages and MoE layers in later ones, with MoE layers—comprising both shared and routed experts—accounting for roughly 90% of total parameters. We quantize all MLP projection weights (`up_proj`, `gate_proj`, `down_proj`) and expert weights to INT8. We apply GPTQ to dynamically update unquantized weights and reduce quantization error. MoE expert activation patterns vary with input data, so we scale the calibration dataset to ensure each expert sees at least $n = 4$ samples

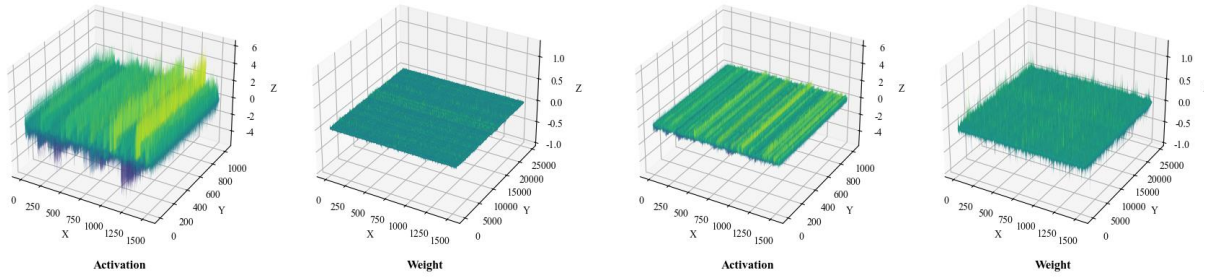


Figure 15 | **Quantization Stats.** *Input activation and weight magnitudes in a DeepSeek-R1 linear layer. The left two subfigures represent the distributions prior to smoothing, and the right two show how smoothing limits extreme values post-processing.*

(typically 40–128) during quantization. To further improve inference efficiency, we fuse the `up_proj` and `gate_proj` operations into a single hardware-accelerated kernel.

Communication Quantization. To lower communication overhead, we quantize expert inputs and outputs to INT8 before sending data between nodes. Additionally, we fuse quantization and dequantization operations directly within the communication operators during inference (see §3.2 for more detail).

KV Cache Quantization. The KV cache in MLA contains both RoPE and non-RoPE components, used to reduce computation during decoding. However, these components significantly increase memory usage for long sequences. To reduce this memory overhead, we quantize non-RoPE components—which show stable numerical distributions—to INT8. For attention layers with lower sensitivity to precision, we further optimize by performing the attention computation entirely in INT8.

5. Transformerless: Towards Fully Disaggregated LLM Serving

We advocate resource disaggregation as a core architectural principle for modern model-serving systems. Disaggregation separates system components into independently scalable units, improves fault isolation, and enables flexible evolution of system software. Network bandwidth is the foundation of disaggregation. A 40 Gb/s scale-out network once enabled memory [3] and OS disaggregation [18] for traditional workloads. Today, 400 GB/s scale-up interconnects make it possible to disaggregate model serving for AI workloads.

To realize these benefits, we introduce **Transformerless**, an architecture that decomposes transformer-based LLMs into modular blocks—attention, feedforward (FFN), and MoE layers—executed independently on NPUs interconnected via high-speed fabric. The hundreds of GB/s UB interconnect in CloudMatrix384 enables aggressive disaggregation without sacrificing inference performance. We envision Transformerless in three stages. First, in disaggregated prefill-decode (§5.1), we isolate compute-bound prefill from memory-bound decode by assigning them to different NPUs. Second, in disaggregated MoE-attention (§5.2), we decouple MoE experts from attention computation, enabling independent scaling by sequence length and batch size. Finally, in dataflow serving at SuperPod scale (§5.3), we aim to break global synchronization altogether, allowing truly independent scaling of each serving component.

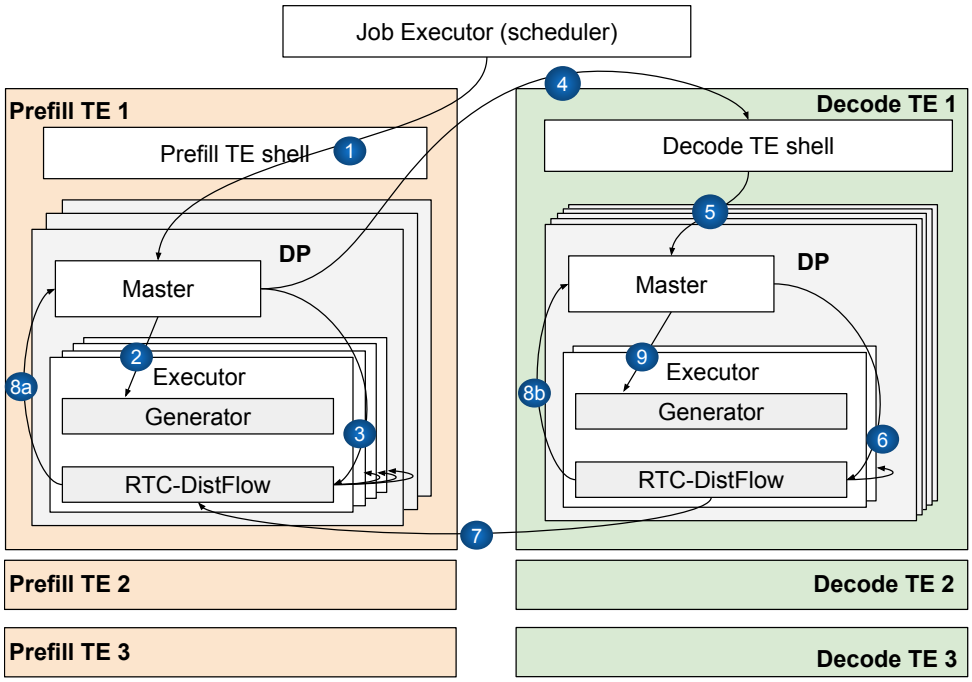


Figure 16 | **The Workflow of Disaggregated Prefill and Decode over CloudMatrix384.** *We support M prefill and N decode deployments with full-mesh connectivity. We illustrate the end-to-end workflow of sending a request from a prefill TE to a decode TE.*

5.1. Disaggregated Prefill and Decode

Disaggregated prefill and decode were concurrently introduced by Splitwise [16], TetriServe [7], and DistServe [24] to reduce contention between the compute-bound prefill and memory-bound decode phases. This design was further refined by Mooncake [17] and MemServe [6].

Despite its adoption, building an efficient and robust disaggregated prefill/decode pipeline remains challenging—especially for large MoE models with expert parallelism—due to two main reasons. First, prefill and decode require distinct parallelization strategies because of differing sequence lengths: we use TP=4 for prefill attention and TP=1 for decode. This necessitates different DP groupings, making colocated execution inefficient. Second, the backend execution diverges: prefill uses an eager graph for dynamic-length support, sequence parallelism, and prefix cache computation; decode uses a static graph for maximum throughput.

We address these challenges with a fully disaggregated pipeline, as shown in Figure 16.

1. A request first arrives at a randomly selected Job Executor (JE), which assigns it to a prefill Task Executor (TE) based on a combination of cache status, system load, and request length. Length-awareness is crucial: co-locating long and short requests in the same DP group leads to stragglers, significantly degrading tail latency and TTFT (§4.3).
2. The prefill TE then schedules the request onto a DP group for computation.
3. Upon completion of prefill, the DP master registers a PD-transfer task with RTC-DistFlow [9]. This task contains only metadata and the addresses of the relevant KV cache blocks; actual transfer is deferred until triggered by the decode phase.
4. Meanwhile, the JE dispatches the request to a decode TE, chosen based on real-time

system load to balance throughput.
5. The decode TE schedules the request to an appropriate DP group via load-aware routing.
6. On the decode side, the DP checks for available KV block slots. If capacity is insufficient, the RECV is deferred, applying backpressure to upstream components. If capacity is sufficient, an asynchronous RECV is submitted to DistFlow. We use the point-to-point send/recv APIs described in §3.1.
7. DistFlow orchestrates the actual KV data transfer, handling all low-level concerns such as SEND/RECV handshakes, ordering guarantees, and TP rank synchronization. Since KV blocks are not self-describing, DistFlow ensures correct semantic pairing between prefill and decode DPs. Each prefill–decode TE pair operates with an isolated DistFlow instance to limit failure domains. However, multiple DistFlow instances can share XCCL buffers to optimize NPU's on-chip memory usage.
8. Each DP polls its DistFlow completion queue. Once transfer completes, the prefill DP releases the KV blocks, while the decode DP enqueues the request for computation.

Heterogeneous Prefill and Decode Deployment. To maximize cost-efficiency, we run prefill and decode on heterogeneous NPUs. Prefill runs on both scale-out Ascend 910 (the second generation) and scale-up CloudMatrix384 NPUs, since it is compute-bound and largely indifferent to interconnect bandwidth. We place decode exclusively on CloudMatrix384, where high-speed interconnect is key to meeting the 35 ms TPOT SLA by accelerating MoE dispatch and combine. When prefill runs on Ascend 910 (the second generation) and decode on CloudMatrix384, we transfer KV cache over RoCE or VPC interconnects, as shown in Figure 2. FlowServe selects the appropriate DistFlow [9] backend based on the network fabric. For MLA models like DeepSeek and Kimi K2, both interconnects satisfy TTFT and TPOT SLAs.

5.2. Disaggregated MoE and Attention

The idea of disaggregating attention from feedforward networks (FFN) and executing them on separate resources was pioneered by works such as FastDecode [5], Lamina [1], and InstAttention [15], and later extended to MoE models by MegaScale-Infer [25]. However, prior work primarily explores small models at limited scale, leaving it unclear whether the same principles hold for large-scale MoE models like DeepSeek.

To bridge this gap, we build the first large-scale disaggregated MoE-Attention system for MoE models over CloudMatrix384. Specifically, we run DeepSeek-V3/R1 with MoE-Attention disaggregated at the decode stage. Our deployment spans a full CloudMatrix384 with 768 NPU dies: 288 dies run EP288 (256 routed experts and 32 shared experts), and 480 dies runs MLA. We use DeepSeek as an example, the same design works for Kimi K2 [11] and Qwen [22].

A key challenge for disaggregated MoE-Attention is maximizing the utilization of MoE NPUs. The MoE component is stateless, and its compute workload scales primarily with batch size, making its execution time predictable. By contrast, MLA computation (such as in DeepSeek) maintains state through KV caches and scales with both batch size and sequence length. This mismatch makes traditional approaches like microbatching or compute-communication overlap insufficient for keeping MoE NPUs fully occupied.

To solve these challenges, we introduce three techniques illustrated in Figures 17 and 18.

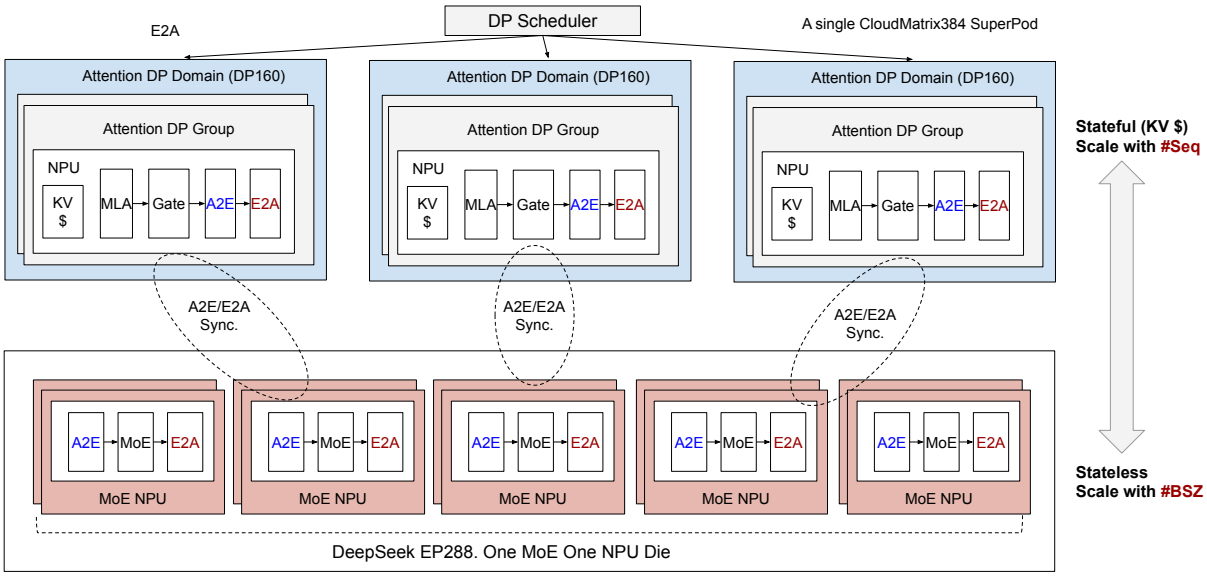


Figure 17 | **The Archiecture of Disaggregated MoE and Attention over CloudMatrix384.** *Our deployment spans a full SuperPod with 768 NPU dies: 288 run EP288 (256 routed experts and 32 shared experts), and 480 handle MLA computation. We deploy three DP domains, each with 160 DP groups and TP=1. Attention DP groups execute full MLA computation, including MLAProlog, Attention, gating, output projection, and A2E/E2A kernels. MoE NPUs run only A2E, MoE computation, and E2A.*

First, we propose new all-to-all communication primitives, `A2E` and `E2A`, tailored specifically for separate MoE and attention deployments (§3.3). A major difficulty at large scale—such as DeepSeek-R1/V3—is asymmetric NPU allocation (e.g., 288 MoE NPUs versus 160 attention NPUs), which renders traditional pull-based mechanisms inefficient. We address this by designing a trampoline forward mechanism, where a subset of MoE NPUs (matching attention NPUs) initially receives data and subsequently forwards it to the remaining MoE NPUs. This two-stage routing reduces metadata overhead and balances bandwidth usage across NPUs, enabling efficient communication at scale. In Figure 18, `A2E` and `E2A` represent the first stage, and `A2E'` and `E2A'` denote the second.

Second, we introduce a new abstraction called **Data Parallel (DP) domain**, which encapsulates multiple DP groups. A system may contain multiple DP domains, but only one domain interacts with MoE NPUs at any given time through `A2E` and `E2A` operations. Without DP domains, all DP groups would communicate concurrently, making microbatching the sole mechanism for overlapping computation and communication. However, excessive microbatching reduces the effective batch size, degrading MoE efficiency. DP domains complement microbatching by enabling parallelism *across* domains (**inter-DP parallelism**), while microbatching enables parallelism *within* a domain (**intra-DP parallelism**). Figure 17 illustrates a deployment with three DP domains, each comprising 160 DP groups (TP=1). Figure 18 shows the corresponding execution timeline, highlighting how DP domains and microbatching (two microbatches per domain, each of size 96) work together to improve throughput and resource utilization.

Third, we propose a **zero-overhead scheduling using persistent kernels** for MoE NPUs. We use three concurrent streams: one for receiving data via `A2E`, one for MoE computation,

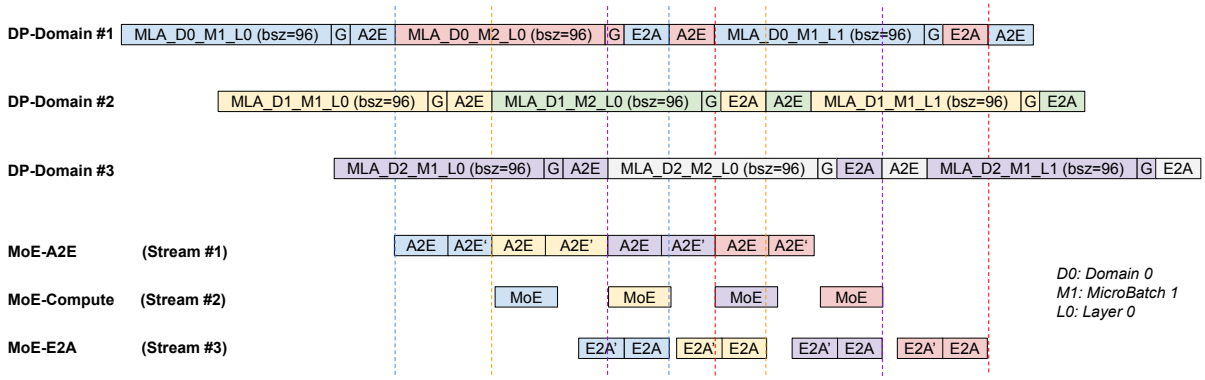


Figure 18 | **The Execution Pipeline of Disaggregated MoE and Attention.** *Note that each DP domain has 160 DP groups with TP set to 1. XCCL A2E and E2A uses two-stage routing. All 288 MoE NPU run that same persistent kernels with three concurrent streams.*

and one for sending data via `E2A`. Each stream runs a persistent kernel in a busy-polling loop without returning to the CPU. This approach is critical because MoE kernels typically execute at microsecond-level granularity, and any CPU interaction (milliseconds) would introduce scheduling delays and degrade overall performance.

5.3. Vision: Dataflow Serving at SuperPod-Scale

No system can truly scale if global synchronizations remain in the data path. While the disaggregated MoE-Attention architecture separates MoE and attention across NPUs, it still relies on two tightly synchronized communication primitives—`A2E` and `E2A`. As a result, a single component failure or straggler can stall the entire system, leaving it vulnerable to cascading delays. We envision the next generation of LLM serving systems to be free of global synchronization, resembling classical dataflow architectures where tensors flow asynchronously between components. Realizing this vision poses several challenges: First, redesigning communication protocols to tolerate latency variation without synchronous barriers. Second, rethinking scheduling and admission control to operate in a decentralized, event-driven manner. Third, ensuring consistency and correctness in the presence of partial results or delayed inputs.

6. Reliability

Ensuring reliability is a key challenge when serving large-scale LLMs over CloudMatrix384, particularly because group communication in MoE can amplify the impact of a single-device failure into a system-wide outage. This section presents our reliability design, focusing on high-level mechanisms. We omit engineering details such as Kubernetes health probes, dependency degradation strategies, and flow control mechanisms for brevity.

6.1. Failure Detection

Timely and accurate fault detection is particularly challenging in xDeepServe, as each FlowServe engine instance may spawn hundreds of processes across tens of nodes. While crash failures are relatively straightforward to identify, detecting *stuck* processes—e.g., due to operators hanging

on group communication—is significantly harder.

To address this, we implement a multi-tiered heartbeat mechanism. The xDeepServe control plane periodically sends heartbeats to each FlowServe engine's TE shell, which in turn forwards heartbeats to the master process of each DP group. These two intervals are decoupled for flexibility. Each DP master runs a single-threaded event loop and only responds to heartbeats when the loop is active. For instance, if an executor hangs before replying to the master, the master's loop stalls and fails to respond—correctly indicating a fault.

However, some failure modes arise outside the main event loop. A common source is the KV-transfer pipeline between the prefill and decode stages. This pipeline operates asynchronously, decoupled from the main loop, and is thus invisible to standard heartbeat checks. Failures in this path, such as a KV cache failing to transfer, are often due to resource saturation rather than transport errors. To detect these issues, we introduce a *link probing* mechanism. By monitoring KV-transfer outcomes and injecting dummy payloads into the channel, we differentiate between decode-side saturation (which delays dummy data) and link-level faults (which block all transmission). This approach enables accurate root cause diagnosis for silent stalls and complements the heartbeat-based fault detection path.

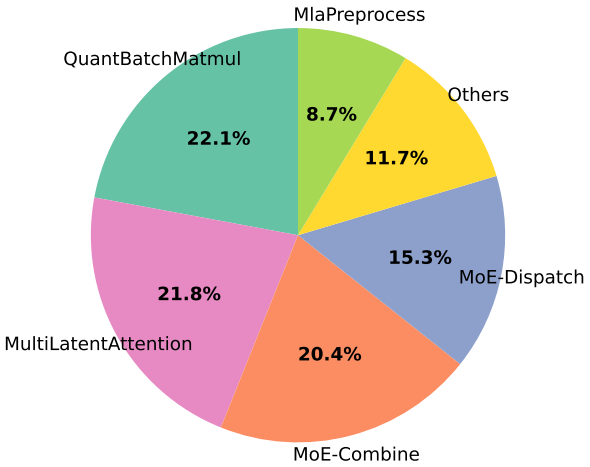
6.2. Failure Recovery

Robust failure recovery is critical for large-scale LLM serving systems, where distributed execution spans hundreds of NPUs across multiple stages. Over time, we evolved our recovery strategies through three key stages—starting with coarse-grained full restarts and progressing toward fine-grained, component-level resilience. This section outlines the evolution of our recovery mechanisms, highlighting trade-offs in complexity, availability, and efficiency.

Stage 1: Restart-the-World. Our initial deployment adopted the simplest strategy: small clusters (e.g., 4 prefill and 1 decode instance, or 4P1D) and full engine restarts upon failure. When an NPU failure is detected, Kubernetes marks the node as tainted, making it unavailable to xDeepServe. To ensure degraded clusters can still run decode instances—which span multiple nodes—we prioritize restarting decode before prefill. While simple and reliable, this approach suffers from two key limitations. First, small clusters underutilize resources due to limited request sharing and pooling. Second, restarting the entire cluster delays recovery; a single prefill failure unnecessarily takes down all associated decode resources.

Stage 2: Prefill/Decode (P/D) Separate Failover. We next moved to shared clusters with multiple prefill and decode instances, improving resource utilization and enabling better workload specialization. For example, some prefill instances can handle long sequences to reduce head-of-line blocking, while decode capacity is pooled across requests. This shift required independent failover logic and introduced challenges in KV cache handling. A key issue is decode fragility—since decode spans many NPUs (e.g., 8 dies), a single failure may idle all others. Early versions of xDeepServe adopted a "**kill-P-to-preserve-D**" policy: prefill instances are terminated to free up resources for restarting decode. Later, we co-designed with EP-LB to support vertical scaling of decode: in failure cases, we reduce the number of DP groups and EP ranks, allowing decode to proceed with fewer NPUs. Each expert maintains at least one replica, while excess replicas are gracefully removed.

Stage 3: Fine-Grained Error Handling. Many transient failures don't require engine



Op	Avg (us)	Min (us)	Max (us)
Dispatch	234	185	1231
Combine	312	165	2939

Figure 19 | **Latency Breakdown for One DeepSeek Decode Iteration.** *Evaluation is conducted on 288 NPU dies with DP288 and EP288. Each die (or DP) uses a batch size of 60. One decode iteration—which includes MTP forward, sampling, main model forward, and final sampling—takes approximately 93 ms. The scheduling bubble between iterations adds around 2 ms. The average MTP acceptance rate is 90% across all DPs. This setup achieves 2400 tokens/s per chip at a TPOT of 50 ms.*

restarts. We focus on two frequent cases: network instability and on-chip memory faults. Network glitches (e.g., from switch flapping or BGP convergence) may temporarily break communication. To tolerate this, FlowServe adopts a *token recomputation* strategy. When certain error codes are detected, all DP groups coordinate a rollback to the previous iteration. A dedicated thread broadcasts the rollback signal to ensure all groups—including those in busy-wait—are notified. The iteration is then re-executed, avoiding full restarts. On-chip memory faults are another common issue. When triggered, we work with the CANN runtime to remap virtual memory and mask the faulty region. While this may cause some KV cache loss and individual request failures, the system remains online and continues serving unaffected requests.

7. Evaluation

7.1. Decode Performance

We evaluate the maximum decoding performance of running DeepSeek [14] on CloudMatrix384.

Setup. We evaluate FlowServe on a single CloudMatrix384 deployment comprising 18 Ascend servers, totaling 288 NPU dies, using a **colocated prefill-and-decode setup**. The system is configured with DP288 and EP288: 256 dies host one routed expert and one redundant expert each, while the remaining 32 dies host one shared expert and one redundant expert each. We enable MTP with one MTP layer. Each DP/die handles a local batch size of 60, yielding a global batch size of $60 \times 288 = 17{,}280$. FlowServe is configured to complete prefill for all requests before entering the decode phase simultaneously. The workload uses fixed 2K-token prompts manually constructed from ShareGPT [19], followed by 2K-token outputs. We enforce fixed-length outputs by enabling `ignore-eos`. Both the MTP module and the main model use greedy sampling during inference.

Performance. We achieve an average TPOT (time per output token) of 50 ms. Each decode iteration—which includes MTP forward, sampling, main model execution, and final sampling (see Figure 13)—takes approximately 93 ms. The scheduling gap between iterations is around 2 ms. With an MTP acceptance rate of about 90% across all DPs, the effective TPOT is computed as $\frac{93+2}{1.9} \approx 50$ ms. This translates to a per-chip throughput of $2 \times 60 \times \frac{1000}{50} = 2400$ tokens/second (two dies per chip, each with batch size 60). For the full DP288 system, the total throughput reaches 345K tokens/second. Figure 19 shows kernel-level latency breakdown for one decode iteration (at approximately 3K sequence length). The attention kernel—scaling with both sequence and batch size—accounts for 21.8% of latency and will grow with longer sequences. Thanks to CloudMatrix384's high-bandwidth UB fabric, `dispatch` and `combine` are efficient, contributing about 36% of the total latency. However, these global synchronization kernels exhibit significant variance: their maximum latency can be up to 10× their minimum. This is because `dispatch` absorbs variance from MLA compute across all DPs, while `combine` absorbs imbalance from MoE compute across experts (see Figure 10 for execution timeline).

Disaggregated MoE and Attention. We evaluate DeepSeek-R1's decoding performance using disaggregated MoE-Attention on a single CloudMatrix384 comprising 768 NPU dies. Prefill is executed separately on other NPU servers. The decode TE is configured as follows: 480 NPU dies are organized into three DP domains, each with 160 DP groups. The remaining 288 NPU dies run EP288. Each DP/die processes a local batch size of 96, resulting in a global batch size of $96 \times 3 \times 160 = 46{,}080$. We use a fixed 2K+2K workload with `ignore-eos` enabled and greedy sampling. Under this setup, we achieve a per-chip throughput of 2400 tokens/second with a 50 ms TPOT. Specifically, `MLAProlog`, `MLA`, `Gating`, and the first stage of `A2E` each take approximately 700 ns per layer. TPOT includes both MTP and main model forwards; the second microbatch of the final layer cannot be overlapped. Accounting for 2 ms scheduler overhead, 5 ms for the MTP layer, and layer-wise compute $(0.7 \times 2 \times 61)$ ms, plus latencies for `A2E` (0.17 ms), `MoE` (0.12 ms), and `E2A` (0.19 ms), the total forward time is approximately 93 ms. Given an MTP acceptance rate of 90% across all DPs, the effective TPOT is $\frac{93}{1.9} \approx 49$ ms.

7.2. Production Workload

We evaluate xDeepServe's performance under a production workload.

A key challenge in production LLM serving is the high variability in input and output lengths, coupled with strict latency requirements. For all DeepSeek and K2 models, we target a TTFT SLA under 2 s and a TPOT SLA of 35 ms in most cases. In particular, for DeepSeek-R1 reasoning models, we support input sequences up to 96K tokens, with up to 32K reasoning tokens and a maximum of 32K output tokens—bounded by the model's 128K context window. Serving long-sequence requests differs fundamentally from short-sequence ones. Long inputs can take up to 30 minutes to process and exhibit increasing latency with sequence length, primarily due to the attention kernel's complexity. To mitigate interference, we allocate dedicated resources to handle these extreme cases, isolating them from short-sequence traffic.

We now report the performance of a representative production setup and workload. The deployment consists of 16 Ascend servers, organized into four prefill TEs and one decode TE. Each prefill TE spans two servers and is configured with DP8 and EP32. The decode TE spans eight servers with DP128 and EP128. The workload features input lengths ranging from 0 to 64K tokens, with an average input length of 13K and an average output length of 2.1K tokens.

Under this setup, we achieve a TTFT of 900 ms and an average TPOT of 34.8 ms.

8. Conclusion

xDeepServe demonstrates how to efficiently serve large-scale MoE models on SuperPod-scale infrastructure through full-system co-design. By disaggregating transformer components, introducing a memory-semantic communication layer, and decentralizing execution with FlowServe, xDeepServe sustains high throughput and low latency across hundreds of NPUs. It runs DeepSeek models in production at 2400 tokens/s/chip while meeting a 50 ms TPOT SLA. Looking ahead, we believe disaggregated execution will become the foundation for future LLM inference systems, especially as models and hardware continue to scale.

9. Contributors

Ao Xiao, Bangzheng He, Baoquan Zhang, Baoxing Huai, Bingji Wang, Bo Wang, Bo Xu, Boyi Hou, Chan Yang, Changhong Liu, Cheng Cui, Chenyu Zhu, Cong Feng, Daohui Wang, Dayun Lin, Duo Zhao, Fengshao Zou, Fu Wang, Gangqiang Zhang, Gengyuan Dan, Guanjie Chen, Guodong Guan, Guodong Yang, Haifeng Li, Haipei Zhu, Haley Li, Hao Feng, Hao Huang, Hao Xu, Hengrui Ma, Hengtao Fan, Hui Liu, Jia Li, Jiang Liu, Jiang Xu, Jie Meng, Jinhan Xin, Junhao Hu, Juwei Chen, Lan Yu, Lanxin Miao, Liang Liu, Linan Jing, Lu Zhou, Meina Han, Mingkun Deng, Mingyu Deng, Naitian Deng, Nizhong Lin, Peihan Zhao, Peng Pan, Pengfei Shen, Ping Li, Qi Zhang, Qian Wang, Qin ZhC Qingrong Xia, Qingyi Zhang, Qunchao Fu, Ren Guo, Ruimin Gao, Shaochun Li, Sheng Long, Shentian Li, Shining Wan, Shuai Shen, Shuangfu Zeng, Shuming Jing, Siqi Yang, Song Zhang, Tao Xu, Tianlin Du, Ting Chen, Wanxu Wu, Wei Jiang, Weinan Tong, Weiwei Chen, Wen Peng, Wenli Zhou, Wenquan Yang, Wenxin Liang, Xiang Liu, Xiaoli Zhou, Xin Jin, Xinyu Duan, Xu Li, Xu Zhang, Xusheng Chen, Yalong Shan, Yang Gan, Yao Lu, Yi Deng, Yi Zheng, Ying Xiong, Yingfei Zheng, Yiyun Zheng, Yizhou Shan, Yong Gao, Yong Zhang, Yongqiang Yang, Yuanjin Gong, Yue Yu, Yuetao Chen, Yukun Zhu, Yulong He, Yusu Zhao, Yuyan Wu, Zenan Zhang, Zhaojin Zhuo, Zhaoyang Ji, Zhefeng Wang, Zheng Wang, Zhenan Fan, Zhenhua Yang, Zhenli Sheng, Zhibin Yu, Zhigang Ji, Zhihao Ren, Zhipeng Bian, Zhixia Liu, Zhiyu Dong, Zhonghua Li, Zhou Yu, Zhuoming Shen, Zhuwei Peng, Zi Ye, Zihao Xiang, Zimin Fu, Zixuan Zhang.

References

[1] Shaoyuan Chen, Wencong Xiao, Yutong Lin, Mingxing Zhang, Yingdi Shan, Jinlei Jiang, Kang Chen, and Yongwei Wu. Efficient heterogeneous large language model decoding with model-attention disaggregation. arXiv preprint arXiv:2405.01814, 2024.

[2] DeepSeek. DeepEP, 2025. Accessed: 2025-07-02.

[3] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 401–414, 2014.

[4] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[5] Jiaao He and Jidong Zhai. Fastdecode: High-throughput gpu-efficient llm serving using heterogeneous pipelines, 2024.

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

[7] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate LLM inference for mixed downstream workloads. CoRR, 2024.

[8] Junhao Hu, Wenrui Huang, Haoyi Wang, Weidong Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. EPIC: efficient position-independent caching for serving large language models. In Proceedings of the 42nd International Conference on Machine Learning, 2025.

[9] Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Baoquan Zhang, Shining Wan, Gengyuan Dan, Zhiyu Dong, Zhihao Ren, Jie Meng, Chao He, Changhong Liu, Tao Xie, Dayun Lin, Qin Zhang, Yue Yu, Hao Feng, Xusheng Chen, and Yizhou Shan. DEEPSERVE: Serverless large language model serving at scale. In Proceedings of the 2025 USENIX Annual Technical Conference, 2025.

[10] Huawei. Ascend C Programming Manual, 2025. Accessed: 2025-07-02.

[11] Kimi. Kimi K2, 2025. Accessed: 2025-07-27.

[12] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In Proceedings of the 29th Symposium on Operating Systems Principles, pages 611–626, 2023.

[13] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. DaVinci: A scalable architecture for neural network computing. In Proceedings of the 2019 IEEE Hot Chips Symposium, pages 1–44, 2019.

[14] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-v3 technical report. CoRR, 2024.

[15] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. Instattention: In-storage attention offloading for cost-effective long-context llm inference. In 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA).

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

[17] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 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, pages 155–170, 2025.

[18] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, 2018.

[19] Sharegpt teams. `https://sharegpt.com/`.

[20] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated {Key-Value} stores. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 33–48, 2020.

[21] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In International Conference on Machine Learning, 2023.

[22] 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. Qwen3 technical report, 2025.

[23] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In Proceedings of the Advances in Neural Information Processing Systems, pages 62557–62583, 2024.

[24] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation, pages 193–210, 2024.

[25] 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. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism.

[26] Pengfei Zuo, Huimin Lin, Junbo Deng, Nan Zou, Xingkun Yang, Yingyu Diao, Weifeng Gao, Ke Xu, Zhangyu Chen, Shirui Lu, et al. Serving large language models on huawei cloudmatrix384. arXiv preprint arXiv:2506.12708, 2025.