Janus: Multi-LLM Serving at Production Scale

Tianbao Zhou[1,*] Yi Wang[2,*] Yu Zhou[3,*] Zirui Liu[1,2,6,*,†] Zhiming Wang[1] Yebo Peng[1]
Yongfu Wang[4] Yi Zhang[1] Jinrun Yin[6] Kemeng Tian[6] Fangcheng Fu[5] Tongxuan Liu[2]
Tao Peng[2] Tong Yang[1] Bin Cui[6] Xupeng Miao[6] Ke Zhang[2]

[1]State Key Laboratory of Multimedia Information Processing, School of Computer Science, Peking University
[2]JD Company [3]University of Chinese Academy of Sciences [4]University of Science and Technology of China [5]Shanghai Jiao Tong University
[6]School of Computer Science & Beijing Key Laboratory of Software and Hardware Cooperative Artificial Intelligence Systems, Peking University

Abstract

Production LLM serving multiplexes hundreds of heterogeneous models on shared clusters, exposing three challenges that existing systems fail to address simultaneously: unpredictable bursts, power-law application popularity, and heterogeneous yet complementary resource demands. We present Janus, a Service–Engine co-designed multi-model serving system built around dual-timescale scheduling. At the Service layer, a Model Scheduler driven by a Performance Oracle combines vector bin-packing every 30 seconds for steady-state colocation with elastic scaling every 0.5 seconds. A new LST-IMH Request Scheduler maximizes SLO attainment with a proven 2-approximation guarantee. At the Engine layer, xTensor virtualizes HBM, while a three-state model lifecycle and device-to-device (D2D) fork enable elastic multi-model colocation and sub-second burst scale-out. We deploy Janus on a 768-device production cluster serving 62 applications and 45.6 M requests/day, and evaluate it with a 603 K-request open-loop replay sampled from that production day. Janus maintains 0.97–1.0 SLO attainment, versus 0.80–0.92 for the strongest baseline, while using 13,440 device-hours/day, 27% less than static ServerlessLLM. Janus is available as open source at https://github.com/Janus2026/Janus.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; *Distributed architectures.*

Keywords: large language model serving, multi-model serving, resource management, autoscaling, HBM virtualization

*Tianbao Zhou, Yi Wang, Yu Zhou, and Zirui Liu contributed equally and are co-first authors. Zirui Liu led the project.
†Corresponding author: Zirui Liu (zirui.liu@pku.edu.cn).

SOSP '26, Prague, Czechia

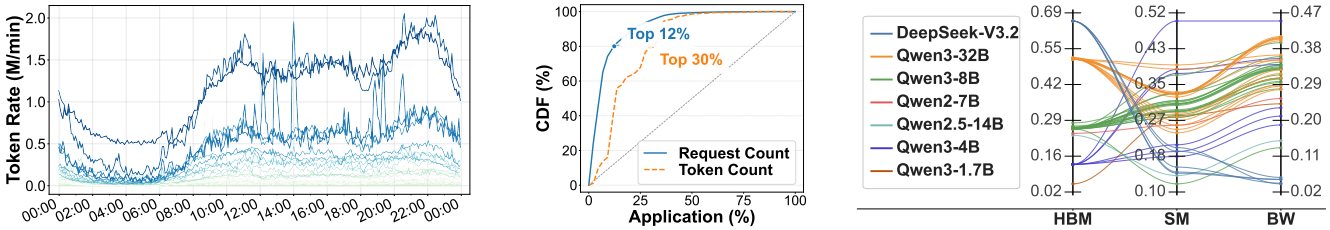
ACM Reference Format:
Tianbao Zhou, Yi Wang, Yu Zhou, Zirui Liu, Zhiming Wang, Yebo Peng, Yongfu Wang, Yi Zhang, Jinrun Yin, Kemeng Tian, Fangcheng Fu, Tongxuan Liu, Tao Peng, Tong Yang, Bin Cui, Xupeng Miao, and Ke Zhang. 2026. Janus: Multi-LLM Serving at Production Scale. In *Proceedings of 32nd ACM Symposium on Operating Systems Principles (SOSP '26).* ACM, New York, NY, USA, 17 pages.

1 Introduction

Large language model (LLM) inference now serves hundreds of heterogeneous models on shared clusters [27, 71]. Modern Model-as-a-Service (MaaS) platforms host models spanning three orders of magnitude in size (from 0.6B-parameter task-specific models to 671B-parameter frontier models) while processing tens of millions of requests daily. Each application has distinct traffic patterns and latency service-level objectives (SLOs) for Time-to-First-Token (TTFT) and Time-per-Output-Token (TPOT). The core challenge is no longer how to maximize the throughput of one model on one machine [24, 69], but how to efficiently multiplex models with diverse request rates, resource footprints, and SLO targets onto a shared cluster.

Our production trace of 62 applications exposes three challenges (§2.2). Second-scale bursts require rapid replica activation. Power-law popularity makes one allocation policy inefficient for both head and tail applications. Heterogeneous demands for HBM capacity (HBM-C), GPU compute (SM), and HBM bandwidth (HBM-BW) create colocation opportunities that scalar device allocation misses.

Existing systems each target a different subset of these challenges and are effective within their intended scope. MuxServe [10] colocates models with multidimensional resource awareness but allocates resources statically, so it does not absorb traffic bursts. ServerlessLLM [11] accelerates model loading via optimized checkpoints but does not target cluster-level elastic scaling or differentiated management for head and tail applications. BlitzScale [72] enables fast per-application scaling but manages each application independently, without cross-application resource coordination. Prism [71] supports multi-model colocation via virtual-to-physical page remapping, but its periodic global re-optimization pauses requests during reconfiguration, limiting its response to sudden bursts. No existing system simultaneously handles bursty traffic, power-law popularity, and multidimensional resource heterogeneity.



(a) 24-hour token rates show bursts. The omitted high-volume class is used in all experiments. **(b)** Request CDF: the top 12% of applications produce 80% of requests. **(c)** Models exhibit complementary HBM, SM, and bandwidth demands.

Figure 1. Production trace analysis of 62 applications on March 10, 2026.

We present Janus, a multi-model LLM serving system for production-grade MaaS clusters. Janus is organized into a centralized *Service* layer making global scheduling decisions and a distributed *Engine* layer managing per-instance resources and model execution. The core design is a *dual-pool resource management* paradigm that schedules at two timescales, loosely analogous to fast and slow thinking [23, 54]. The analogy describes scheduling cadence, not request execution. A *Steady Pool* runs the slow path: it tightly colocates tail applications to minimize device usage through resource complementarity. An *Elastic Pool* runs the fast path: it provides sub-second autoscaling for head applications to absorb traffic bursts.

The Service layer makes scheduling decisions with three components. A *Model Scheduler* guided by a Gaussian-process-based *Performance Oracle* operates on two paths: a slow path runs vector bin-packing for model colocation every 30 seconds. A fast path triggers elastic model scaling every 0.5 seconds. The *Request Scheduler* employs LST-IMH, a new algorithm that formalizes multi-instance request dispatching as a parallel-machine deadline scheduling problem. LST-IMH achieves a 2-approximation guarantee for maximizing on-time completions, generalizing the classical Moore–Hodgson algorithm, an optimal single-machine deadline scheduler [32], to the multi-machine setting.

The Engine layer restructures the inference runtime to support multi-model colocation and fast scaling. An *xTensor* (cross-model tensor) manager virtualizes HBM. xTensor is a contiguous virtual HBM address space. Its manager organizes all physical pages into a unified pool, supporting bidirectional allocation (weights growing upward, KV caches/activations growing downward) and eliminating static memory partitioning. A three-state lifecycle (active/warm/cold) decouples model lifecycle from instance lifecycle, enabling two reactivation paths: host-to-device (H2D) wakeup for routine recovery and device-to-device (D2D) *fork*, pulling weights directly from a peer device for sub-second burst scale-out.

On a 768-device supernode, we replay 603 K requests sampled from a 45.6 M-request production day covering 62 applications. Janus maintains 0.97–1.0 SLO attainment, versus 0.80–0.92 for the strongest baseline, while using 27% fewer device-hours than static ServerlessLLM.

In summary, we make the following contributions:

1. Production traces reveal three concurrent challenges: unpredictable bursts, power-law application popularity, and heterogeneous, complementary resource demands. We design Janus as a Service–Engine co-designed MaaS system to address them jointly (§2–3).
2. The Service layer combines Performance-Oracle-guided dual-timescale model scheduling with LST-IMH, a request scheduler with a 2-approximation guarantee (§4).
3. The Engine layer combines xTensor virtual HBM, a three-state model lifecycle, and D2D fork to support elastic multi-model colocation and sub-second scale-out (§5).
4. We deploy Janus on a 768-device cluster serving 62 applications and 45.6 M requests/day. A 603 K-request trace replay demonstrates 0.97–1.0 SLO attainment and 27% fewer device-hours than static ServerlessLLM (§7). Janus is available as open source [20].

2 Background and Motivation

2.1 Background

LLM inference. LLM inference is autoregressive. Each request starts with a *prefill* phase that processes the entire input prompt in parallel and is compute-intensive. The subsequent *decode* phase generates tokens one at a time. Each step reads the full model weights and all previously computed context, making it memory-bandwidth-bound [41]. To avoid redundant computation, serving systems store per-token key–value (KV) caches and reuse them across decode steps [24, 69]. Recent systems further disaggregate prefill and decode onto specialized instances (P/D disaggregation) [2, 40, 52, 58, 78], enabling independent scaling.

Model-as-a-Service. Cloud platforms increasingly offer Model-as-a-Service (MaaS): a shared cluster hosts hundreds of models from different applications, each with its own traffic pattern and latency requirements [5, 11, 16, 27, 71, 72]. The standard service-level objectives are Time-to-First-Token (TTFT) for the prefill phase and Time-per-Output-Token (TPOT) for decode. *SLO attainment*, the fraction of requests meeting both targets, is the primary metric. Device usage is secondary.

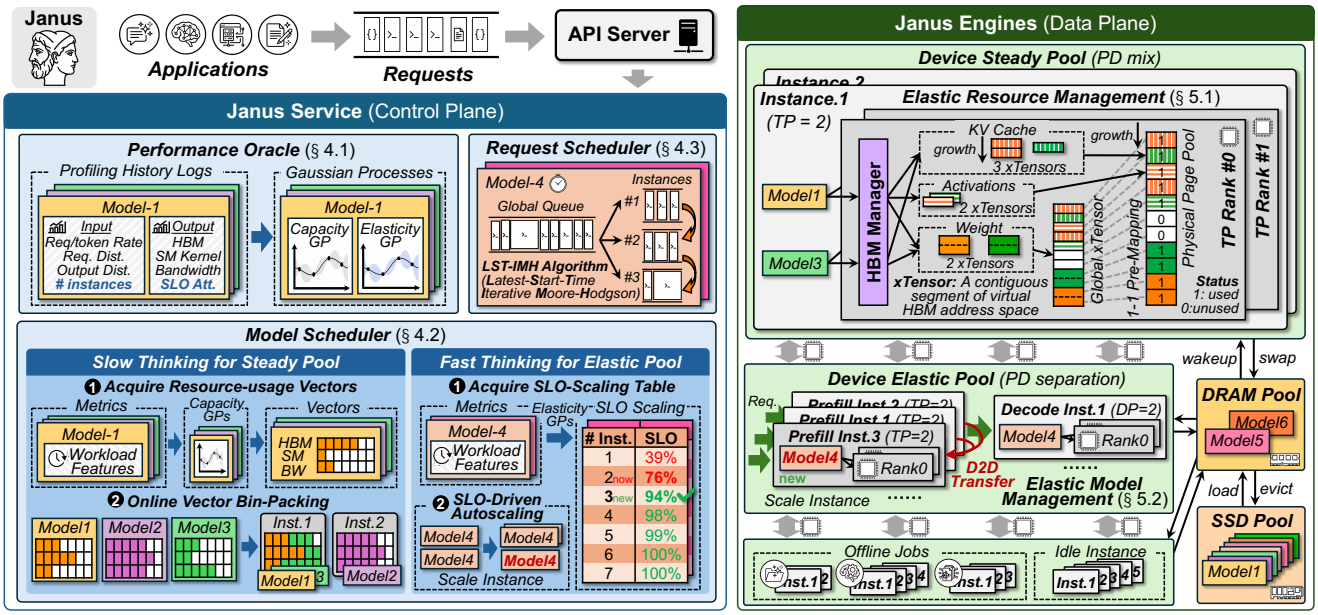


Figure 2. Janus system architecture. The Service layer maintains a Performance Oracle that feeds the Model Scheduler's slow and fast paths. The Request Scheduler dispatches requests to the provisioned instances. The Engine separates resources into Steady and Elastic Pools for colocation and on-demand scaling.

Deployment setting. Janus targets a *supernode* cluster in which all devices are interconnected via a fully connected high-speed fabric. The aggregate device-to-device (D2D) bandwidth within a supernode is significantly higher than the host-to-device (H2D) path, making D2D the preferred channel for weight transfer. Under this topology a 32B-parameter model with TP = 2 (~32 GB per device in FP16) can be activated via D2D fork in 783 ms (Table 1). Similar topologies exist on commodity platforms (e.g., an NVLink domain within a DGX node [34]). We report concrete bandwidth numbers on our target hardware in §7.1.

2.2 Motivation

We analyze one day of production traces (March 10, 2026) from our MaaS cluster serving 62 applications and highlight three observations that shape the design of Janus.

Observation 1: Traffic is bursty and unpredictable. Figure 1a plots the per-application token rate over a 24-hour window. Most applications exhibit relatively stable traffic. However, a small fraction experiences sudden bursts: request rates surge by 5–10× within seconds [57], driven by promotional events or upstream pipeline triggers that are invisible to the serving system. These bursts leave no time for proactive scaling. Conventional autoscaling mechanisms require full instance initialization (container startup, memory allocation, weight loading, and collective-group establishment), incurring tens of seconds to minutes of delay [3, 11, 22, 72]. Even on high-bandwidth interconnects, the H2D weight-loading step alone takes 0.4–2.7 s (Table 1), and conventional systems add process-level cold-start overhead on top of that. *The system must be able to activate new model replicas within seconds to keep pace with bursty traffic.*

Observation 2: Application popularity is heavily skewed. Figure 1b shows the cumulative distribution of request and token volume across applications. The top 12% of applications account for 80% of all requests, and the top 30% account for 80% of all tokens. Head applications demand higher model-parallel degrees [34] to reserve sufficient HBM for KV caches. Tail applications each receive only a handful of requests per minute, yet collectively they represent significant model diversity that the platform must support. Their absolute traffic volume is relatively stable. *Head and tail applications have fundamentally different resource needs and cannot be managed with a single policy [10, 27].*

Observation 3: Resource demands are heterogeneous and complementary. Figure 1c profiles representative applications along three resource dimensions: HBM capacity (dominated by weights and KV caches), compute throughput, and memory bandwidth. Different applications stress different dimensions. Large-parameter models are HBM-heavy, small high-throughput models saturate compute, and decode-intensive workloads are bandwidth-bound. Crucially, these profiles are largely *complementary*: a memory-heavy model and a compute-heavy model can share a device with minimal mutual interference, because they contend on different hardware resources [10, 71].

Mainstream engines adopt a one-instance-one-model architecture [24, 35, 77] and therefore cannot exploit this complementarity. Recent work has begun to explore multi-model colocation [10, 27, 71] but treats resources as a single scalar rather than a multidimensional vector. Combined with varying tensor-parallelism degrees, this means that placement decisions must consider all resource axes jointly rather than treating device count as a single scalar. Prior work exploits the complementarity between prefill and decode on a single model [10, 78]. Our setting generalizes this to hundreds of heterogeneous applications whose resource profiles span a much wider space.

3 System Overview

The production observations define three design requirements: (1) sub-second elastic scaling for unpredictable bursts, (2) differentiated management for head and tail applications, and (3) resource-aware placement that exploits multidimensional complementarity across colocated models. Figure 2 presents the architecture of Janus. Before describing each component, we explain how the three design requirements from §2 shape the overall structure.

From requirements to design. The multi-model scheduling problem decomposes into three subproblems, *placement*, *scaling*, and *routing*, that must be solved jointly. Placement bounds per-instance capacity, scaling adjusts it to demand, and routing must respect both. If solved independently (*e.g.* by routing to an instance whose capacity a stale placement no longer supports), these decisions can break SLOs under bursts. Resource complementarity turns placement into a vector bin-packing problem over HBM-C, SM, and HBM-BW, requiring a *Performance Oracle* that learns per-model resource profiles via Gaussian processes. Power-law popularity calls for differentiated management: a *Steady Pool* colocates tail applications via bin-packing on the slow path, while an *Elastic Pool* provides rapidly scalable instances for head applications on the fast path. Bursty applications are pre-isolated into the Elastic Pool because a slow-path repack cannot absorb second-scale bursts. Pool assignment follows volume and variability (Figure 1a). Bursty traffic demands sub-second scaling. The fast path queries the Oracle's *Elasticity GP* to determine replica counts proactively, and the Engine provides *D2D fork* for sub-second model activation. LST-IMH has a 2-approximation guarantee.

The resulting system has a centralized *Service* layer for global scheduling (Oracle, Model Scheduler, Request Scheduler) and a distributed *Engine* layer for per-instance resource management and model execution.

Janus Service (§4). The Service layer coordinates all Engine instances through three components. (1) The *Performance Oracle* (§4.1) exposes two GP families: a *Capacity GP* that maps workload features to a 3D resource vector (HBM-C, SM, HBM-BW) for placement, and an *Elasticity GP* that predicts SLO attainment as a function of workload and instance count for scaling. (2) The *Model Scheduler* (§4.2) consumes the Oracle's predictions and operates on two paths. The slow path solves vector bin-packing for Steady Pool placement, and the fast path autoscales Elastic Pool instances. (3) The *Request Scheduler* (§4.3) uses LST-IMH to maximize on-time completions.

Janus Engine (§5). Instances are partitioned into two pools, one per Model Scheduler path: the Steady Pool serves the slow path and the Elastic Pool serves the fast path. The *Steady Pool* colocates multiple models per instance. *Elastic resource management* (§5.1) virtualizes HBM via xTensor and multiplexes compute via multi-stream execution, enabling dynamic resource redistribution among colocated models. The *Elastic Pool* dedicates each instance to a single model. An *elastic model manager* (§5.2) maintains a three-state lifecycle (active/warm/cold) and provides H2D wakeup and D2D fork for sub-second scaling, while preserving compatibility with per-model optimizations (§5.3).

Putting it all together. The two layers form a closed feedback loop. The Performance Oracle continuously refines its GP models from runtime telemetry collected by the Engine. When workload patterns shift, the Model Scheduler reacts on the appropriate path, rebalancing colocation via the slow path or scaling instances via the fast path, and the Engine executes these decisions through its elastic resource and model management mechanisms. We describe the Service and Engine layers in detail next.

4 Janus Service

4.1 Performance Oracle

The Performance Oracle uses two per-model *Gaussian process* (GP) surrogate families [45] to drive the Model Scheduler. The two are non-overlapping: the *Capacity GP* maps workload features to a resource vector (slow path), while the *Elasticity GP* maps workload features plus instance count to SLO attainment (fast path). They share input features but predict different targets and never model the same quantity. GPs are data-efficient under sparse observations, which are common for new models. After a one-time $O(n^3)$ factorization over n profiling points, they support $O(n)$ prediction and incremental Sherman–Morrison updates [50] as workloads drift. In plain terms, the Sherman–Morrison formula updates the fitted inverse matrix after a new observation without refitting the GP from scratch.

Capacity GP. The Capacity GP serves the slow path. It predicts a normalized steady-state resource-demand vector $\mathbf{v} = (v_{\text{HBM-C}}, v_{\text{SM}}, v_{\text{HBM-BW}}) \in [0, 1]^3$. The components are the fractions of one device's HBM capacity, GPU compute throughput, and HBM bandwidth required under the current workload. The input features for each model are drawn from a recent sliding window: token rate λ_t, request rate λ_r,

mean and variance of input lengths ($\mu_{\text{in}}, \sigma^2_{\text{in}}$), and mean and variance of output lengths ($\mu_{\text{out}}, \sigma^2_{\text{out}}$). The Model Scheduler feeds $\mathbf{v}$ into the Steady Pool's bin-packing algorithm (§4.2).

We represent length distributions as their mean and variance rather than histograms: these statistics are low-dimensional, stable across window sizes, and capture the distributional shape that most affects resource consumption. A normalized mutual information (NMI) analysis on production traces in the online supplement further validates this choice [21].

Elasticity GP. The Elasticity GP serves the fast path. It predicts SLO attainment (*e.g.* TTFT on-time rate) from the same workload features plus the current instance count m. Given a target (*e.g.* 95%), binary search over m yields the minimum instance count for the Elastic Pool (§4.2).

Including m as an input is necessary because scaling is nonlinear: the Request Scheduler's LST-IMH algorithm (§4.3) exploits heterogeneity in request lengths, so adding instances yields superlinear throughput gains when input lengths are skewed. The GP learns this interaction from runtime data. Its uncertainty estimate causes the system to overprovision when observations are scarce.

Data collection. Offline profiling bootstraps both GPs before a model goes online. Once serving, the Oracle updates incrementally from Engine telemetry.

4.2 Model Scheduler

The Model Scheduler consumes the Oracle's predictions and operates on two paths aligned with the dual-timescale paradigm. On the *slow path*, which runs every 30 seconds, it queries the Capacity GP to obtain each model's resource vector and solves an online vector bin-packing problem to colocate models onto the fewest Steady Pool instances. On the *fast path*, which runs every 0.5 seconds, it queries the Elasticity GP to determine the minimum instance count that meets the SLO target and scales Elastic Pool instances accordingly. The 0.5-s interval balances burst responsiveness against control-plane overhead. Varying it by ±50% does not change the qualitative result.

4.2.1 Slow Path: Bin-Packing. The slow path colocates models onto Steady Pool instances. Models with *complementary* resource profiles, one heavy on HBM and another on compute, can share a device with little interference. We formalize this as an online 3D vector bin-packing problem [14, 33] and design a potential-based algorithm that incrementally adjusts placements rather than recomputing the global assignment from scratch. Our algorithm has two properties: (1) it greedily pairs complementary models on arrival, and (2) it triggers repacking when a bin's balance deteriorates past a threshold, rather than on every model departure, keeping model migrations infrequent.

We model each device instance as a bin with normalized capacity $(1, 1, 1)$ across HBM-C, SM, and HBM-BW. Each model i has a demand vector $\mathbf{v}_i \in [0, 1]^3$ produced by the Capacity GP. A packing is feasible when $\sum_{i \in B} v_{i,k} \leq 1$ for every dimension k and every bin B. The goal is to minimize the total number of bins, freeing the remaining instances for the Elastic Pool, while also minimizing model migrations.

Both the insertion rule and the repacking trigger are derived from a single *fragmentation potential* Φ that quantifies how unevenly a bin's resources are consumed (formal definition in the online supplement [21]). On insertion, we place each new model in the feasible bin whose residual resources best complement it (*e.g.* a bin skewed toward bandwidth scores highest for a bandwidth-heavy model), minimizing the one-step potential increase. If no feasible bin exists, a new bin is opened. On departure, we track a *damage ratio* ρ that measures how much pairwise complementarity has been lost. A repack is triggered when ρ exceeds a threshold α. A background thread also runs a global first-fit decreasing consolidation every 30 seconds to reclaim instances for the Elastic Pool. The online supplement provides a mean-field analysis [21] showing that (1) each model experiences only $O(1)$ repacks over its lifetime, and (2) the bin-count overhead is close to 1, confirming near-optimal packing. Because insertions are incremental and repacks are $O(1)$ per model, the scheduler scales to the hundreds of models in production without global re-solving.

4.2.2 Fast Path: Autoscaling. The fast path adjusts the number of Elastic Pool instances for each model every 0.5 seconds. The loop is a discrete controller rather than an ad hoc heuristic: the Elasticity GP is the plant model mapping instance count to SLO attainment, binary search finds the minimal feasible actuation (*i.e.* fewest instances meeting the target), and the anti-oscillation window acts as a hysteresis band that suppresses thrashing.

Target computation. The Elastic Pool's device budget B is the number of instances left after the Steady Pool has been provisioned. The Steady Pool is prioritized because losing a Steady Pool instance forces eviction of all colocated models, whereas the Elastic Pool can gracefully degrade by scaling down replicas. Given B, the scheduler queries the Elasticity GP with current workload features to find the minimum instance count m for each model that meets the SLO target. If the sum of targets exceeds B, targets are scaled down proportionally: model i's quota becomes $q_i = m_i \cdot B / \sum_j m_j$. Because q_i is generally fractional while instance counts must be integral, we round quotas via the Hamilton largest remainder method [4], so each allocation differs from its proportional share by at most one instance.

Anti-oscillation. A sliding window stores the two most recent scaling plans. A new plan is accepted only if it differs from *both* previous plans. Otherwise it is discarded as oscillation. This eliminates thrashing when demand hovers near a scaling threshold, *e.g.* alternating between 2 and 3 replicas on successive arrivals.

Overlapped execution. Scaling overlaps: instances released by shrinking models are immediately assigned to growing ones, whose reactivation overlaps the old model's drain and halves transition latency.

4.3 Request Scheduler

The Request Scheduler dispatches pending requests to the instances provisioned by the Model Scheduler. Its objective is to maximize the number of requests that meet their SLO deadlines. The core algorithm, LST-IMH, operates on the Elastic Pool's multiple prefill instances and specializes gracefully to the single-instance case in the Steady Pool.

Dispatch architecture. The Service maintains a centralized request queue per model and tracks each prefill instance's Estimated Prefill Done Time (EPDT). Rather than per-instance RPC pulls, the Service simulates pulls locally. When any instance crosses a 20-ms EPDT threshold, it runs LST-IMH over the full backlog and pushes requests to idle instances, exploiting a cross-instance global view for better orderings than local queues [71]. The 20-ms trigger balances waiting long enough to form a useful global ordering against leaving an instance idle. The result is likewise insensitive to a ±50% change. Completed prefills go to the decode instance with the smallest backlog.

Problem formulation. Consider a model with m prefill instances whose availability times are $T_1, \ldots, T_m$ (each T_i is the instance's current EPDT). The backlog contains n requests, each with an estimated processing time p_j (from a TTFT predictor) and a deadline $d_j = a_j + \mathrm{TTFT}_{\mathrm{SLO}}$, where a_j is the arrival time. The goal is to maximize the number of on-time completions, $\max \sum_j (1 - U_j)$ where $U_j = \mathbf{1}[C_j > d_j]$. In scheduling-theory notation this is $P_m \mid T_i \mid \sum U_j$, a generalization of $P_m \| \sum U_j$ which is NP-hard.

LST-IMH algorithm. We propose *LST-IMH* (Latest-Start-Time Iterative Moore–Hodgson), a new scheduling algorithm for the parallel-machine deadline problem. While the classical Moore–Hodgson algorithm [32] is optimal for a single machine, the multi-machine setting $P_m \mid T_i \mid \sum U_j$ is NP-hard. LST-IMH gives a practical 2-approximation by combining a machine-ordering strategy with iterative single-machine reductions. The key idea is to schedule the most constrained machine first: sorting instances by decreasing T_i prevents lightly loaded instances from consuming jobs that only a heavily loaded instance could still finish on time. On each machine, we solve a shifted single-machine problem with effective deadlines $d_j^{(i)} = d_j - T_i$ via Moore–Hodgson, which maximizes on-time jobs optimally in $O(n \log n)$. Jobs rejected by any machine's Moore–Hodgson pass are returned to the backlog for subsequent machines. Jobs still unassigned after the last machine are shed with an immediate timeout callback so that the application can execute its degradation logic. The full pseudocode and complexity analysis are given in the online supplement [21]. The total running time is $O(mn \log n)$, sub-millisecond in practice.

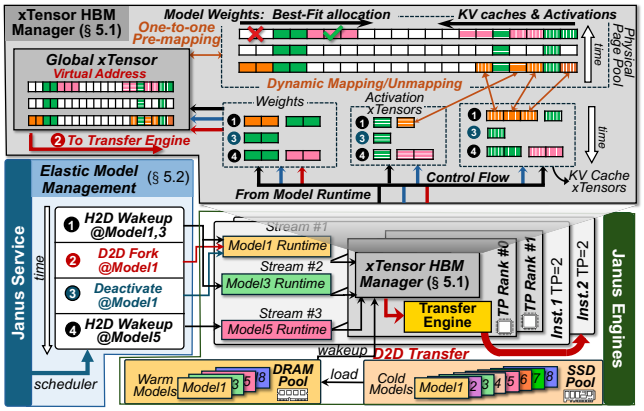


Figure 3. Janus Engine architecture. xTensor (§5.1) unifies HBM management for weights and runtime state. Elastic Model Management (§5.2) controls lifecycle transitions and model wakeups/forks.

Guarantee. LST-IMH is a 2-approximation: it always returns at least half the optimal number of on-time jobs. The proof appears in the online supplement [21]. The bound holds for any machine ordering, but the LST ordering is critical in practice: an instance with less slack is harder to schedule, so prioritizing it is near-optimal in practice.

Request shedding. Because Moore–Hodgson rejects any job whose processing time would cause other jobs to miss their deadlines, requests that no instance can accommodate are naturally shed rather than queued indefinitely. This provides principled admission control: sacrificing a small number of expensive (long-prompt) requests maximizes the overall on-time rate. Alternatively, shed requests can be re-enqueued with a relaxed deadline (reentry mode), trading slightly lower attainment for a lower failure rate.

5 Janus Engine

5.1 Elastic Resource Management

Figure 3 illustrates the goal of elastic resource management: supporting multi-model colocation with dynamic HBM and compute redistribution across models. Conventional engines bind memory allocation, execution contexts, and scheduling state to a single model, preventing fine-grained HBM sharing and turning resource adjustments into expensive cross-instance operations.

The key insight is to *virtualize* HBM: decouple physical pages from any particular model and manage them in a single pool. Any free page can be claimed by whichever model needs it next, eliminating static partitioning.

xTensor abstraction. Janus virtualizes HBM through *xTensor*, a contiguous virtual HBM address space whose manager interposes beneath the default device-memory

allocator of deep-learning runtimes (*e.g.* PyTorch [39], TensorFlow [1]). Upper-layer code still manipulates weights, KV caches, and activations as ordinary tensors. The xTensor manager controls only how their virtual addresses are backed by physical HBM pages. At instance startup, the xTensor manager acquires handles for all available HBM pages at the hardware's minimum page granularity and organizes them into a *unified physical page pool*, implemented as a mutex-protected double-ended queue for concurrent allocation and reclamation. In the initial state, tensors are created with virtual addresses only, and their physical bindings are deferred until use.

Why separate weight and KV management. Virtualized HBM incurs map/unmap overhead (37/235 μs per 2 MB page), reaching seconds for large models with naive page-by-page mapping. Janus therefore handles weights and KV/activation tensors differently: weights occupy large, stable contiguous regions, whereas KV caches are fine-grained and dynamic.

Weight management. For weights, Janus constructs at startup a *global xTensor*: a virtual region sized to match the entire physical page pool whose physical pages are mapped once and registered with the communication backend. Subsequent weight operations (allocation, deallocation, and migration) reduce to reserving or releasing contiguous segments within this space, without triggering further driver-level mapping. To reduce interference with short-lived allocations, weight segments are drawn from the high-numbered end of the unified physical page pool. Janus uses a best-fit policy for segment allocation to preserve large contiguous free regions and mitigate fragmentation. When fragmentation prevents allocating a sufficiently long contiguous segment in the global xTensor space despite sufficient free HBM overall, Janus falls back to allocating a separate virtual region and explicitly mapping the required pages.

KV cache and activation management. KV caches are managed at page granularity, but Janus keeps driver-level mapping off the common request-serving path. Each model maintains a cache of already-mapped pages. New KV blocks are allocated from this cache on the fast path, and only cache depletion triggers a background worker to map fresh physical pages from the global pool. Released KV blocks return to the mapped-page cache for immediate reuse. KV caches and activations acquire pages from the low-numbered end of the unified physical page pool, separating short-lived fine-grained allocations from long-lived weight segments. Activation tensors follow the same discipline.

Compute multiplexing. Beyond HBM elasticity, Janus multiplexes compute resources across models via multi-stream parallel execution. Each colocated model maintains an independent stream context. The hardware scheduler interleaves kernels from different streams. Unlike isolation schemes that rely on chip-specific features such as MPS [36] or MIG [37], this mechanism requires no vendor-specific capability and therefore generalizes across hardware platforms. Together, HBM elasticity and compute multiplexing allow multiple models to share a single instance efficiently, improving overall device utilization.

5.2 Elastic Model Management

The shared resource pool allows each colocated model to choose its own tensor- and data-parallel degrees at creation time, so large and small models can coexist efficiently. Janus currently fixes these degrees at creation time: it scales replica count and changes colocation, but does not recompile or repartition a live model. Dynamic parallelism reconfiguration is orthogonal and left to future work.

The key insight is to *decouple model lifecycle from instance lifecycle*. Creating a new model only sets up a lightweight runtime context while reusing the instance-level resource pool. To avoid idle models permanently occupying resources while still supporting low-latency reactivation, Janus introduces a three-state lifecycle (active/warm/cold) that controls how much resource each model retains at different idle levels.

Three-state lifecycle. Each model has three states. *Active*: weights reside in HBM, and the model serves online requests directly. *Warm*: weights are offloaded to host pinned memory while the collective-communication group, runtime metadata, and framework context are retained, supporting fast recovery in the Steady Pool. *Cold*: the weights, KV pages, and collective-communication group are all released. Only minimal framework state remains, providing long-term standby with minimal footprint. Reactivation reloads weights from persistent storage.

Model initialization. The first time a model is placed on an instance, the Engine initializes its runtime context (collective-communication group, runtime metadata, and framework context) while reusing the instance-level physical page pool (§5.1). This initialization happens once per model per instance and can materialize the model directly as active, warm, or cold. Later lifecycle transitions reuse the same context.

Deactivation. As shown in ③ of Figure 3, when a model transitions from active to warm or cold, Janus first drains in-flight requests to ensure no kernel still references the model's memory. It then *unmaps* the model's KV cache pages and marks the corresponding weight segment in the global xTensor as free, immediately returning the physical pages to the shared physical page pool.

The model then follows one of two residency paths. For *active-to-warm*, Janus ensures a weight copy exists in contiguous host-pinned DRAM for fast wakeup. If such a copy already exists it is reused. Otherwise, the weights are restored from persistent storage into host-pinned memory. The collective-communication group, runtime metadata,

and framework context are all preserved. For *active-to-cold*, Janus additionally releases the DRAM weight copy, retaining only the minimal state needed for future reconstruction. Because the global xTensor is established at startup (§5.1), subsequent reactivation requires no page-level remapping: Janus simply locates a free weight segment in the global xTensor and restores the weight data into it.

Reactivation. Janus provides two reactivation paths.

H2D wakeup. As shown in ①④ of Figure 3, this path recovers warm models in the Steady Pool. Their weights reside in contiguous host-pinned blocks. Janus asynchronously transfers large per-layer chunks to HBM while overlapping transfer with subsequent execution, keeping the transfer count and overhead low.

D2D fork. As shown in ② of Figure 3, this path handles burst scale-out in the Elastic Pool. Inter-device links (*e.g.* NVLink) offer higher bandwidth than host-mediated H2D. Direct device-to-device transfer is therefore preferable for urgent scaling. We call this path *D2D fork*: analogous to `fork()`, it creates a new model replica by pulling weights directly from an active copy on another device. Janus reuses the global xTensor (§5.1) as a communication buffer. Each active model reports the offsets of its weights within the global xTensor to the Service layer. On fork, the Service sends the offsets to the target instance, which pulls weights directly from the source device without host-memory staging. The two paths, H2D wakeup for recovery and D2D fork for burst scaling, balance availability and scale-out speed.

5.3 Framework Compatibility

Janus preserves framework execution logic and restructures only the underlying resource layer. Each model receives an independent runtime context at initialization (§5.2), so per-model optimizations such as prefix caching, chunked prefill, scheduling–computation overlap, and speculative decoding continue to work unmodified. Only prefill–decode (P/D) disaggregation requires adaptation because it pre-registers KV cache addresses with the communication backend. Under virtual HBM management, a freshly allocated KV block has no physical page binding, so the standard registration call fails. Janus resolves this by registering the global xTensor at startup and translating block virtual addresses to offsets (online supplement [21]).

6 Implementation

We implement Janus on top of an existing production-grade inference framework, adding ∼8 KLOC of C++ to the Service layer (12 KLOC total) and ∼21 KLOC of C++ to the Engine layer (170 KLOC total). The Service layer handles model registration, instance state tracking, and resource-view aggregation in addition to the scheduling logic described in §4. The two layers communicate via `brpc` for control-plane coordination, while bulk data movement for P/D disaggregation and remote weight transfer is carried by Mooncake [44], which serves as Janus's D2D transfer engine. Our implementation extends five key paths (instance management, memory management, model loading, hibernation/wakeup, and remote weight transfer) while reusing the base framework's executor, collective-communication, and model-runtime infrastructure. Although our current deployment uses a specific accelerator platform, Janus's core mechanisms require only device-memory mapping, asynchronous data transfer, and streaming execution, making them portable across compatible runtimes.

7 Evaluation

7.1 Experimental Setup

Hardware. We evaluate Janus on a CloudMatrix 384 supernode [18, 79] comprising 48 10U Atlas 800T A3 servers connected by 56 LingQu 630 V1 switches running Huawei's UnifiedBus 1.0 interconnect. Each server integrates four Kunpeng 920 CPUs and eight Ascend 910C NPU modules. Each NPU module contains two dies, each die acting as an independent device with 376 TFLOPS and 64 GB HBM. Thus, 48 servers × 8 modules × 2 devices yields 768 devices in total. Each node provides 3 TB DRAM and shares a unified address space with HBM via UnifiedBus. The proprietary UnifiedBus fabric uses seven independent planes and connects all 384 NPU modules across the supernode. The vendor's 784 GB/s figure is aggregate bidirectional bandwidth between modules. We report the per-device, one-way bandwidth relevant to a D2D fork below. Within the supernode, D2D links deliver 163.6 GB/s (spec 196 GB/s), H2D links deliver 59.0 GB/s (spec 140 GB/s), and per-device HBM bandwidth is 1490 GB/s (spec 1600 GB/s). Unlike node-local fast interconnects (*e.g.* NVLink), this fabric lets Janus source a fork from another server within the supernode. Its measured D2D bandwidth is 2.8× H2D.

Experimental environment. The W0 end-to-end experiment runs on the full supernode. Component experiments use one or two servers (16 or 32 devices under our per-die counting). Janus uses its xLLM-based Engine with the Ascend runtime, an etcd-backed Service control plane, and the Mooncake transfer engine for D2D fork. In production, each model is provisioned at the tensor-parallel degree matching its size (TP = 1 for models below 14B, TP = 2 for 14B/32B, and TP = 16 for DeepSeek-V3), and this same configuration is used for W0 and all baseline comparisons. Component microbenchmarks explicitly sweep TP where noted. For a controlled comparison, all systems receive the same device pool, model checkpoints, tensor-parallel degrees, SLO targets, and ordered request stream. Placement, scaling, and memory configurations are described under "Baselines."

Workload. We use eight workloads, with additional details in the online supplement [21]. *W0* is derived from

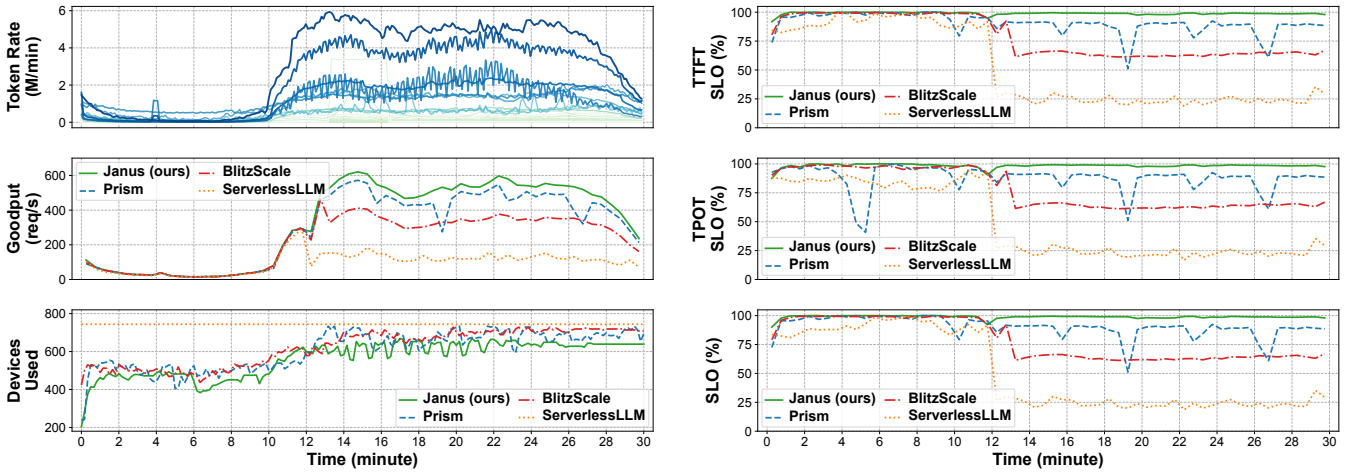


Figure 4. End-to-end performance over time across 62 applications (one line per application in the workload panel): goodput, devices used, TTFT SLO attainment, TPOT SLO attainment, and overall SLO attainment.

one full day of production traffic (March 10, 2026) covering 62 online applications, 9 models (Qwen3-0.6B to Qwen3-32B [64], Qwen2.5-3B/14B [66], Qwen2-7B [65], and DeepSeek-V3 [9]). We construct W0 in two steps. First, we proportionally sample the original 45.6 M requests to ~603 K, retaining application identity, request order, input/output lengths, and relative timestamps. This bounds the experiment's request count. We then compress the 24-hour timestamp axis to 30 minutes, counteracting the lower arrival intensity caused by sampling while keeping the replay practical. The resulting workload runs at about 0.64× the production day's average arrival rate and preserves per-application traffic shares and normalized burst shapes. Each application is a fine-tuned variant of one base model. Fine-tuning changes only the weights, so applications sharing the same base model have identical resource profiles (Figure 1c). Traffic follows a diurnal pattern with peak token rates of 5–6 M/min. *W0* is used for the end-to-end experiment (§7.2). *W1* samples representative head and tail applications, compressed into 120 s (§7.4). *W2–W7* use three public datasets [8, 57, 76] for trace generality (§7.3).

Replay protocol. We replay W0 open-loop: requests are issued at their compressed timestamps without waiting for earlier completions, so every system receives the same arrival sequence. Latency covers client submission through streamed token delivery, and errors and timeouts count as SLO misses.

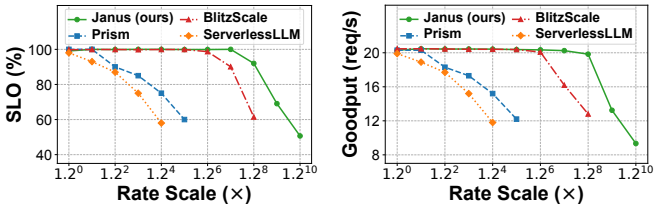
Baselines. We compare against three state-of-the-art systems. *ServerlessLLM* [11] accelerates model loading via optimized checkpoint formats but lacks cluster-level elastic scaling. We partition the device pool evenly and statically bind each application to a fixed partition, so ServerlessLLM occupies all available devices throughout the experiment. *BlitzScale* [72] enables layer-granularity progressive loading so new instances can serve before weights are fully loaded. We use its native per-application autoscaler, which scales each application independently without cross-application resource coordination. *Prism* [71] supports multi-model colocation via virtual-to-physical page remapping and two-level scheduling but lacks cluster-level scaling. We provision the same fixed 768-device pool as ServerlessLLM. Prism performs its native periodic global multi-model placement and remapping within that fixed pool, but cannot reallocate application-level device budgets at runtime.

Metrics. We report five metrics. *TTFT SLO attainment [71]*: the fraction of requests whose time-to-first-token meets the SLO target. *TPOT SLO attainment [71]*: the fraction of requests whose time-per-output-token meets the SLO target. *SLO attainment [61]*: evaluated at per-token granularity. For a request generating n tokens, token i is on-time if its delivery time $T_i \leq \text{TTFT\_SLO} + (i-1) \times \text{TPOT\_SLO}$. The metric is the ratio of on-time tokens to total tokens. Failed requests count all tokens as SLO misses. *Goodput [74]*: the handled request rate (req/s), counting only requests that meet the end-to-end SLO. *Devices used*: the active device count.

7.2 End-to-End Performance

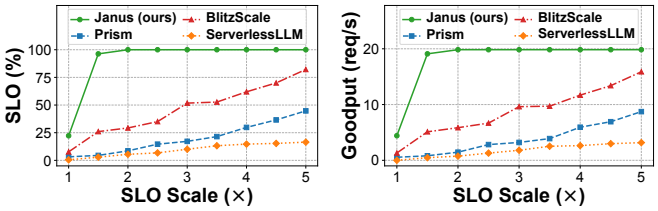
We replay the production-derived trace (*W0*: 62 applications, many with bursty traffic) across all four systems (Figure 4). The first ~650 s corresponds to nighttime low traffic. The daytime peak follows with a sharp traffic surge.

SLO attainment. Janus maintains 0.97–1.0 SLO attainment throughout the entire trace, including the daytime peak (≥0.98). It is the only system that sustains near-perfect attainment across both periods. BlitzScale performs adequately at night (~96.0%) but drops to 0.60–0.65 during the day: high-traffic applications are starved by numerous low-traffic ones competing for devices. Prism fluctuates between



(a) SLO attainment. (b) Goodput.

Figure 5. Rate scale sensitivity: SLO attainment and goodput as the request rate increases from 1× to 1.2^{10}× (≈ 6.19×).



(a) SLO attainment. (b) Goodput.

Figure 6. SLO scale sensitivity: SLO attainment and goodput as the SLO target tightens.

0.80 and 0.92 during the day and dips to 0.51 around 1150 s. Its periodic repacking causes repeated brief service disruptions. ServerlessLLM collapses to 0.22–0.30 once daytime traffic arrives: nearly three-quarters of requests violate their SLOs. TTFT and TPOT attainment follow similar trends.

Goodput. All four systems are similar at night (30–80 req/s). During the day, Janus leads at 400–500 req/s (peak ~620). Prism reaches 300–400 req/s but drops during repacking. BlitzScale (~150) and ServerlessLLM (50–150) trail because static partitions starve head applications.

Device usage. Janus is the most resource-efficient: it scales between 390–480 (night) and 580–660 (day) devices, consuming 13,440 device-hours/day. ServerlessLLM statically locks all 768 devices. BlitzScale (~500–720) and Prism (600–740, oscillating) use more on average. Janus saves 27% over ServerlessLLM (18,432 device-hours/day) while delivering far higher SLO attainment.

Global coordination. To isolate the contribution of cross-application coordination from the underlying colocation and scaling mechanisms, we compare four configurations on *W0*, reporting per-application mean and minimum SLO attainment. Although Prism already coordinates periodic placement within a fixed device pool, augmenting it with Janus's runtime cross-application device allocator (*Prism-with-global*) lifts its mean from 92.7% to 97.6%, confirming that global allocation is a large part of the gain. Its minimum stays low (50.2% → 57.2%) because Prism's periodic remapping cannot react to bursts. Conversely, removing global coordination from Janus (*Janus-no-global*) drops the mean only slightly (99.4% → 96.9%) yet the minimum falls to 82.0%, since D2D fork and vector packing still protect most applications. Only full Janus sustains both a high mean (99.4%) and a high worst case (90.0%): global coordination raises the average, while D2D fork and multidimensional packing defend the burst-time tail. This attributes the end-to-end gain to distinct mechanisms and shows the baseline is treated fairly: Prism benefits from the same runtime device allocator when granted one.

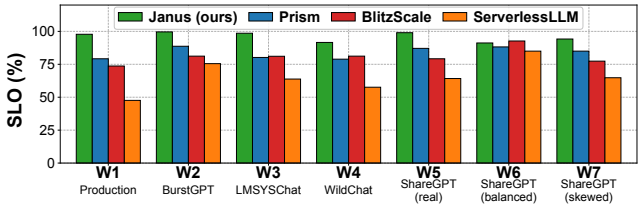


Figure 7. SLO attainment of Janus and baselines across seven traces (*W1–W7*).

7.3 Sensitivity Analysis

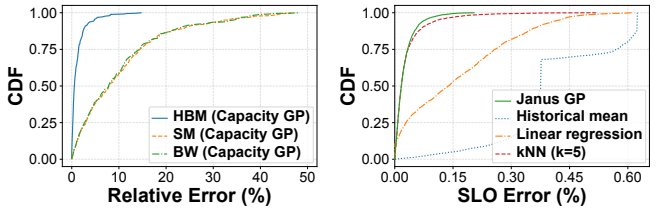
Rate scale. We sweep the request rate from 1× to 1.2^{10}× (≈ 6.19×) (Figure 5). At the same SLO target (≥90%), Janus sustains 1.3–1.5× the request rate of BlitzScale and over 3× that of Prism and ServerlessLLM. Fast-path head-model replication and slow-path tail-model colocation provide the coordination absent from baselines.

SLO scale. We tighten the SLO threshold to evaluate robustness (Figure 6). At the same attainment level (≥90%), Janus tolerates an SLO threshold 5× tighter than BlitzScale. The gap over Prism and ServerlessLLM is even larger.

Trace generality. We evaluate SLO attainment across the production-sampled *W1* as well as six additional workloads (*W2–W7*) derived from public datasets (Figure 7). Janus achieves over 90% SLO attainment on all traces and outperforms every baseline in most cases. The advantage is larger on bursty, skewed traces where sub-second elastic scaling and differentiated pool management take effect. On traces with relatively uniform request distributions (e.g., *W6–W7*, Poisson arrivals), the gap narrows because all systems face similar pressure. Note that this experiment runs on a smaller device set with lower request rates than the end-to-end evaluation. Under these conditions Prism's colocation strategy is more effective, allowing it to outperform BlitzScale.

7.4 Component Analysis

Latency breakdown. Figure 9 decomposes per-request latency at request rate scales from 1× to 5×. At 1×, average latency is 529 ms with prefill and decode accounting for 96.3%.



(a) Capacity GP error. **(b)** GP versus simpler predictors.

Figure 8. Oracle accuracy: (a) Capacity GP error. (b) Elasticity GP versus simpler predictors.

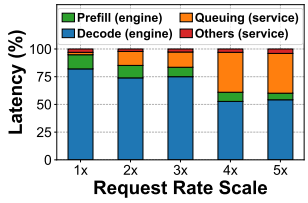


Figure 9. Latency breakdown versus request rate.

Table 1. Lifecycle latency (ms): deactivation, H2D wakeup, and D2D fork.

TP / Model	Deact.	H2D	D2D
2 / Qwen3-8B	307	460	408
2 / Qwen3-32B	524	1147	783
4 / Qwen3-8B	313	387	388
4 / Qwen3-32B	524	885	698
16 / DeepSeek-V3	1278	2712	742

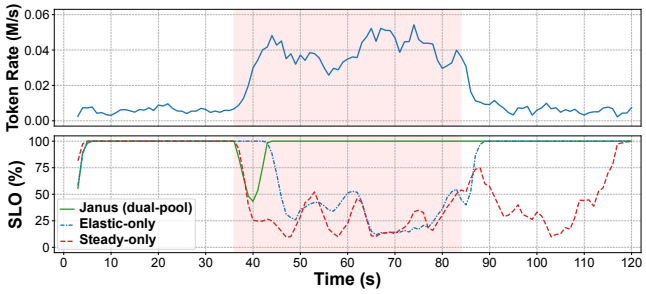
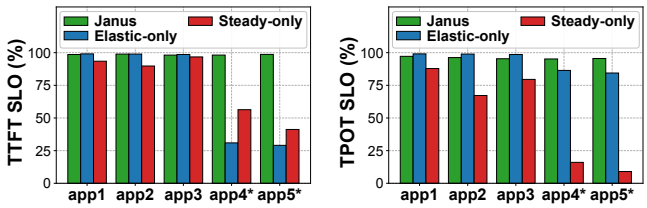


Figure 10. Overall SLO attainment over a 2-minute trace segment under three pool designs.



(a) TTFT SLO attainment. **(b)** TPOT SLO attainment.

Figure 11. Per-application TTFT/TPOT SLO attainment under three pool designs for five representative applications.

At 5×, average latency grows to 1928 ms (3.6×), while prefill and decode remain relatively stable (84–238 ms and 425–1270 ms, respectively). The increase is dominated by queuing, which rises from 3.4% at 1× to 43.3% at 5× (835 ms), indicating that the scheduling pipeline becomes the bottleneck under heavy load. Other overheads (tokenization, dispatch, CPU processing) remain negligible throughout (1–11 ms).

Absolute latency. Beyond SLO-attainment ratios, we report absolute user-visible latency over the full *W0* trace. At P50/P90/P99, TTFT is 57.5/161/395 ms and TPOT is 7.2/12.3/18.5 ms, well within the SLO targets even at the tail, confirming that the high attainment reflects genuinely low latency rather than loose SLOs.

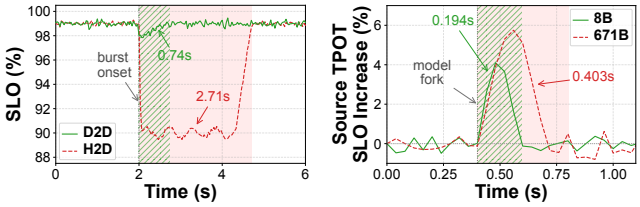
Performance Oracle. We evaluate the Gaussian process (GP) prediction accuracy with an 80/20 train/test split (Figure 8a). The Capacity GP predicts per-model HBM, compute, and bandwidth utilization. HBM prediction is the most accurate: median error 0.54%, P90 2.75%, P99 9.71%. Compute and bandwidth predictions are less precise but acceptable: median errors of 7.86% and 7.84%, P90 errors of 24.18% and 23.24%, respectively. The Elasticity GP predicts SLO attainment given the current workload and instance count. Its error CDF is shown in Figure 8b (Janus GP). Its median absolute error is 2.5 percentage points, with P80 and P90 errors of 9.2 and 19.0 percentage points, respectively. This accuracy is sufficient for the Model Scheduler to trigger timely scale-up and scale-down.

Oracle vs. simpler predictors. Would a simpler predictor suffice? We compare the GP against a historical mean, linear regression, and kNN (k=5) under identical features and split (Figure 8b). The historical mean fails as demand shifts, and linear regression collapses on the nonlinear targets. kNN is the strongest baseline, but the GP matches or beats it everywhere and leads on the SLO tail. In the cold-start regime (8–50 training samples), the GP is roughly 2× more accurate than kNN, exactly when a newly onboarded model gives the scheduler few observations. The GP is also the only predictor with calibrated uncertainty, which drives conservative over-provisioning under sparse data.

Pool partitioning. We ablate Janus's dual-pool design by comparing it against two single-pool variants: elastic-only and steady-only. The experiment runs for 120 s on five applications, where app4* and app5* are bursty with a traffic spike during 36–84 s.

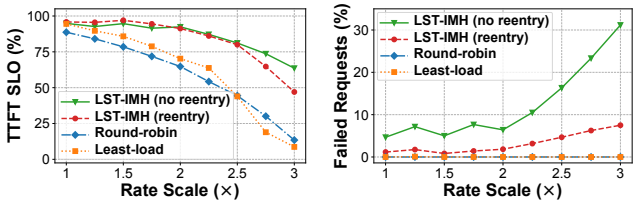
Dual-pool achieves 97.8% overall SLO attainment (Figure 10): SLO briefly drops at burst onset but recovers within 9 s via sub-second Elastic Pool activation. Elastic-only reaches only 59.0% and steady-only 37.1%, as neither can simultaneously protect steady applications and absorb bursts. Per-application results (Figure 11) confirm that dual-pool keeps all five applications above 96% TTFT/TPOT, unlike either single-pool variant.

Model lifecycle latency. Table 1 shows the latency of three lifecycle operations across different model sizes and tensor-parallelism degrees. D2D fork achieves sub-second activation for all configurations: 388–783 ms for Qwen3-8B/32B and 742 ms for DeepSeek-671B. In contrast, H2D wakeup takes 387–1147 ms for Qwen3 models and 2712 ms



(a) 671B burst. **(b)** Source interference.

Figure 12. D2D fork evaluation. (a) SLO attainment during a 671B-parameter DeepSeek-V3 (TP = 16) burst. (b) Source-instance P99 TPOT increase for 671B and 8B models.



(a) TTFT SLO attainment. **(b)** Failed requests.

Figure 13. Request scheduling ablation: TTFT SLO attainment and failed-request ratio under different scheduling policies.

for DeepSeek-671B, 3.7× slower than D2D. This sub-second D2D scaling enables Janus's Elastic Pool to absorb traffic bursts without violating SLOs. Deactivation latency (307–1278 ms) is determined by model depth and KV cache size, independent of weight size.

Table 1 isolates transfer latency. Figure 12a shows the largest-model end-to-end effect. During a DeepSeek-V3 (671B, TP = 16) burst, D2D fork reactivates a replica in 0.74 s and holds SLO at ∼98%, whereas H2D-only takes 2.71 s and dips to ∼90%. Forking only mildly perturbs the source instance (Figure 12b): P99 TPOT rises at most 5.7% for 671B (42.6-ms baseline, 403-ms transfer) and 4.0% for 8B (18.3-ms baseline, 194-ms transfer) and recovers immediately.

Request scheduling. We compare four scheduling policies at 1×–3× request rate scales, with the baseline rate set high enough that the system is near saturation at 1× (Figure 13).

LST-IMH consistently outperforms Round-Robin and Least-Load. At 2×, LST-IMH (reentry) achieves 91.2% TTFT attainment and LST-IMH (no-reentry) 92.4%, versus 64.8% for Round-Robin and 70.2% for Least-Load. At 3×, LST-IMH (reentry/no-reentry) maintains 47.0%/63.6%, while Round-Robin and Least-Load drop to 13.4% and 9.2%. The no-reentry variant yields higher attainment by shedding requests that cannot meet their SLOs, reserving resources for feasible ones. The reentry variant accepts more requests by re-enqueuing rejected ones with relaxed deadlines, lowering the failure rate at the cost of slightly lower attainment.

Bin-packing quality. Over a 24-hour trace (Figure 14), Janus's online incremental heuristic matches the OR-Tools exact bin count on 99.99% of the trace, never more than one bin worse, while deciding 4.3× faster (0.30 vs. 1.28 ms). It updates placements incrementally as models arrive and depart, rather than re-solving a global ILP/SAT on every change, retaining near-optimal quality at a fraction of the cost.

xTensor efficiency. We isolate the xTensor manager with eight microbenchmarks. Building weights on the pre-registered global xTensor is 414×–4427× faster than naive page-by-page mapping (the naive 1 GB case takes 17.5 ms), which is what makes single-instance multi-model residency and D2D fork practical. The KV fast path avoids driver mapping and is 675× faster than the slow path (50 ns vs. 33.7 μs). Raw map/unmap costs 37/235 μs per 2 MB page, fragmentation overhead is only 1.07×, and P/D address translation takes ∼30 ns for an isolated block lookup and 3.4 μs for a batched 512-block request. These results confirm the unified pool imposes negligible steady-state overhead while enabling the fast weight and KV paths the rest of the system depends on. **xTensor end-to-end ablation.** To attribute colocation directly to xTensor, we disable it and fall back to static HBM partitioning (one model per process). In a residency-density experiment with 12 services over 3 base models, the shared xTensor pool needs one device versus three for both static partitioning and per-process xTensor (3× fewer). In a separate end-to-end experiment, four 1.7B models share one device under staggered bursts. Neither configuration runs out of memory (Figure 15, left/middle). In that experiment, removing xTensor reduces mean SLO attainment to 81% and pushes P99 TTFT to several seconds, whereas dynamic HBM redistribution maintains 100% SLO attainment and P99 TTFTs of tens of milliseconds. xTensor is thus a precondition for dense, burst-tolerant colocation, not merely an optimization.

xTensor under churn. Finally, we stress the unified pool with sustained KV allocation churn with cycles of 10, 30, and 60 seconds (Figure 15, right). Fragmentation stays bounded below 0.3% throughout, with no runaway growth, confirming stability under sustained allocation churn.

8 Related Work

Multi-LLM serving. AlpaServe [27] statistically multiplexes model variants, while INFaaS [46] selects variants to satisfy latency targets. S-LoRA [47], dLoRA [60], and Punica [6] serve LoRA adapters by sharing a base model. Full-weight systems must also place distinct weights: MuxServe [10] multiplexes them, DeltaZip [67] compresses model deltas, Prism [71] remaps pages, and Aegaeon [61] pools GPUs at token granularity. WarmServe [30] and Torpor [70] reduce activation costs. ServerlessLLM [11] accelerates cold starts through multi-tier checkpoint storage and

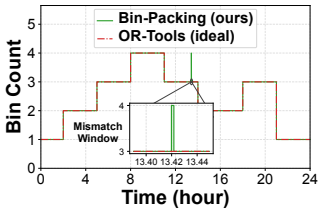


Figure 14. Incremental bin-packing versus OR-Tools optimum over 24 h.

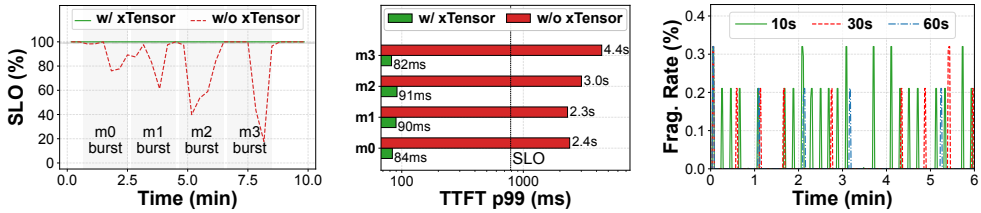


Figure 15. xTensor evaluation. Left/middle: end-to-end ablation of four 1.7B models (*m0–m3*) on one device under staggered bursts, with and without xTensor. Right: fragmentation stays bounded (<0.3%) under 10/30/60 s churn cycles.

locality-aware placement. Janus jointly manages bursty head applications and a stable tail, colocating low-volume full-weight models while retaining isolated replicas for fast scale-out during unpredictable bursts.

SLO-aware scheduling. ORCA [69] introduces continuous batching, FastServe [59] uses preemption, and ExeGPT [38] searches schedules under latency constraints. Llumnix [53] migrates requests to rebalance load. Cocktail [15] and Orion [51] coordinate multiple models, Andes [28] models user-visible quality of experience, and JIT-Serve [74] handles uncertain request lengths. VTC [48] provides token-based fairness across clients. Janus instead dispatches one model's backlog across provisioned instances with a 2-approximation in $O(mn \log n)$ time. The formulation includes heterogeneous instance availability times, which represent when each replica can begin its next request after current work completes.

Elastic scaling and model switching. Prior work dynamically resizes training jobs [19, 43, 56, 62]. LLM serving additionally preserves request and KV-cache state. Llumnix [53] and LoongServe [58] migrate requests or adjust parallelism. SpotServe [31] adapts to preemptible capacity. BlitzScale [72], DeepServe [16], and WarmServe [30] accelerate activation. DeepServe [17] and HydraServe [29] specialize serverless loading, while Weaver [13] reduces the resident footprint through attention offloading. HFX [68] also uses device-to-device transfer, but serves tasks over a shared model and manages weights and KV state separately. Janus applies D2D fork to heterogeneous full-weight models within its dual-pool controller. Request migration remains complementary: it rebalances one model's instance set, while the controller reallocates devices across applications as demand changes.

Device memory management. vLLM [24] pages KV caches, SGLang [77] reuses prefixes, and vAttention [42] and vTensor [63] apply virtual memory to dynamic tensor allocation. Other systems evict, compress, reuse, or offload KV state [12, 25, 26, 49, 75]. PQCache [73] and RetroInfer [7] recast long-context KV access as retrieval rather than retaining the full cache. Prism [71] remaps pages for weights and KV caches but not activations, while Aqua [55] offloads context to remote-GPU HBM. Most of these designs manage KV state separately from model weights. xTensor unifies weights, KV caches, and activations in a bidirectional pool whose pre-registered global region also buffers D2D forks.

9 Conclusion

We presented Janus, a multi-model LLM serving system that co-designs a centralized Service layer with a distributed Engine layer around dual-timescale scheduling. The Service layer combines slow-path vector bin-packing, fast-path sub-second scaling, and LST-IMH request scheduling with a 2-approximation guarantee. The Engine integrates xTensor virtual HBM, a three-state model lifecycle, and D2D fork for elastic colocation and burst scale-out. On a 768-device supernode serving 62 applications and 45.6 million requests per day, a 603K-request production-derived replay achieves 0.97–1.0 SLO attainment with 27% fewer device-hours than ServerlessLLM. Dual-pool control reaches 97.8% SLO attainment, versus 59.0% and 37.1% for elastic-only and steady-only operation. D2D fork reactivates a 671B replica in 0.74 s instead of 2.71 s via H2D, while xTensor triples model residency density and keeps fragmentation below 0.3% under sustained churn. Component ablations show that the Steady and Elastic Pools are complementary: neither single-pool variant simultaneously protects steady traffic and absorbs bursts. Together, these mechanisms combine dense tail-model colocation with sub-second head-model scale-out under production bursts.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (2024YFB2906603), in part by the National Natural Science Foundation of China (62372009 and U23B2048), the Beijing Municipal Natural Science Foundation (L2603008), and the Fundamental Research Funds for the Central Universities, Peking University. We thank the anonymous reviewers and shepherd for their constructive feedback.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 7, 18 pages.

[3] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: fast pipelined context switching for deep learning applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 28, 16 pages.

[4] Michel L. Balinski and H. Peyton Young. 2010. *Fair Representation: Meeting the Ideal of One Man, One Vote* (2nd ed.). Brookings Institution Press.

[5] Zinuo Cai, Zebin Chen, Ruhui Ma, and Haibing Guan. 2024. SMSS: Stateful Model Serving in Metaverse With Serverless Computing and GPU Sharing. *IEEE Journal on Selected Areas in Communications* 42, 3 (2024), 799–811. doi:10.1109/JSAC.2023.3345401

[6] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2024. Punica: Multi-Tenant LoRA Serving. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 1–13. https://proceedings.mlsys.org/paper_files/paper/2024/file/054de805fcceb78a201f5e9d53c85908-Paper-Conference.pdf

[7] Yaoqi Chen, Jinkai Zhang, Baotong Lu, Qianxi Zhang, Chengruidong Zhang, Jingjia Luo, Di Liu, Huiqiang Jiang, Qi Chen, Jing Liu, et al. 2025. Retroinfer: A vector-storage approach for scalable long-context llm inference. *arXiv preprint arXiv:2505.02922* (2025).

[8] Wei-Lin Chiang, Zhuohan Li, Ziqing Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E Gonzalez, et al. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. *See https://vicuna.lmsys.org (accessed 14 April 2023)* 2, 3 (2023), 6.

[9] DeepSeek-AI, Aixin Liu, Bei Feng, et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[10] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. 2024. MuxServe: flexible spatial-temporal multiplexing for multiple LLM serving. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 473, 13 pages.

[11] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: low-latency serverless inference for large language models. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 8, 19 pages.

[12] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, 111–126.

[13] Shiwei Gao, Qing Wang, Shaoxun Zeng, Youyou Lu, and Jiwu Shu. 2025. Weaver: Efficient Multi-LLM Serving with Attention Offloading. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, 587–595. https://www.usenix.org/conference/atc25/presentation/gao

[14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 455–466. doi:10.1145/2619239.2626334

[15] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2022. Cocktail: A Multidimensional Optimization for Model Serving in Cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1041–1057. https://www.usenix.org/conference/nsdi22/presentation/gunasekaran

[16] Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Jie Meng, Baoquan Zhang, Shining Wan, Gengyuan Dan, Zhiyu Dong, Zhihao Ren, Changhong Liu, Tao Xie, Dayun Lin, Qin Zhang, Yue Yu, Hao Feng, Xusheng Chen, and Yizhou Shan. 2025. DeepServe: Serverless Large Language Model Serving at Scale. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC)*.

[17] Junhao Hu, Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Jie Meng, Baoquan Zhang, Shining Wan, Gengyuan Dan, Zhiyu Dong, Zhihao Ren, Changhong Liu, Tao Xie, Dayun Lin, Qin Zhang, Yue Yu, Hao Feng, Xusheng Chen, and Yizhou Shan. 2025. DEEPSERVE: Serverless Large Language Model Serving at Scale. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association. https://www.usenix.org/conference/atc25/presentation/hu-junhao

[18] Huawei Technologies Co., Ltd. 2026. *Atlas 800T A3 Supernode Technical White Paper* (document version 07 ed.). Huawei Technologies Co., Ltd. Product architecture and technical specifications.

[19] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 721–739. https://www.usenix.org/conference/nsdi21/presentation/hwang

[20] Janus Authors. 2026. Janus: Multi-LLM Serving at Production Scale (Source Code). https://github.com/Janus2026/Janus. Open-source repository.

[21] Janus Authors. 2026. Janus: Supplementary Materials. https://github.com/Janus2026/Janus/blob/main/supplementary_materials.pdf.

[22] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. 2025. Serverless Cold Starts and Where to Find Them. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 938–953. doi:10.1145/3689031.3696073

[23] Daniel Kahneman. 2011. *Thinking, Fast and Slow.* Farrar, Straus and Giroux, New York.

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[25] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: efficient generative inference of large language models with dynamic KV cache management. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 9, 18 pages.

[26] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM knows what you are looking for before generation. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 722, 24 pages.

[27] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[28] Jiachen Liu, Jae-Won Chung, Zhiyu Wu, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. 2024. Andes: Defining and Enhancing Quality-of-Experience in LLM-Based Text Streaming Services. *arXiv e-prints* (2024), arXiv–2404.

[29] Chiheng Lou, Sheng Qi, Chao Jin, Dapeng Nie, Haoran Yang, Yu Ding, Xuanzhe Liu, and Xin Jin. 2026. HydraServe: Minimizing Cold Start Latency for Serverless LLM Serving in Public Clouds. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association. https://www.usenix.org/conference/nsdi26/presentation/lou

[30] Chiheng Lou, Sheng Qi, Rui Kang, Yong Zhang, Chen Sun, Pengcheng Wang, Bingyang Liu, Xuanzhe Liu, and Xin Jin. 2025. WarmServe: Enabling One-for-Many GPU Prewarming for Multi-LLM Serving. *arXiv preprint arXiv:2512.09472* (2025).

[31] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 1112–1127.

[32] J. Michael Moore. 1968. An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Manage. Sci.* 15, 1 (Sept. 1968), 102–109. doi:10.1287/mnsc.15.1.102

[33] Lars Nagel, Nikolay Popov, Tim Süß, and Ze Wang. 2023. Analysis of Heuristics for Vector Scheduling and Vector Bin Packing. In *Learning and Intelligent Optimization: 17th International Conference, LION 17, Nice, France, June 4–8, 2023, Revised Selected Papers* (Nice, France). Springer-Verlag, Berlin, Heidelberg, 583–598. doi:10.1007/978-3-031-44505-7_39

[34] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. doi:10.1145/3458817.3476209

[35] NVIDIA. 2024. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[36] NVIDIA. 2026. *Multi-Process Service*. NVIDIA. https://docs.nvidia.com/deploy/mps/ Version v590, accessed 2026-04-01.

[37] NVIDIA. 2026. *NVIDIA Multi-Instance GPU User Guide*. NVIDIA. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/ Accessed 2026-04-01.

[38] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. 2024. ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 369–384. doi:10.1145/3620665.3640383

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA.

[40] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2025. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture* (Buenos Aires, Argentina) *(ISCA '24)*. IEEE Press, 118–132. doi:10.1109/ISCA59077.2024.00019

[41] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curan, 606–624. https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cdfb45e3d06dbfca2780-Paper-mlsys2023.pdf

[42] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1133–1150. doi:10.1145/3669940.3707256

[43] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 1–18. https://www.usenix.org/conference/osdi21/presentation/qiao

[44] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[45] {Carl Edward} Rasmussen and {Christopher K. I.} Williams. 2006. *Gaussian processes for machine learning*. Vol. 2. MIT Press, Uited States.

[46] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[47] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. 2024. SLoRA: Scalable Serving of Thousands of LoRA Adapters. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 296–311. https://proceedings.mlsys.org/paper_files/paper/2024/file/906419cd502575b617cc489a1a696a67-Paper-Conference.pdf

[48] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024. Fairness in serving large language models. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 52, 24 pages.

[49] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: high-throughput generative inference of

large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1288, 23 pages.

[50] Jack Sherman and Winifred J Morrison. 1950. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics* 21, 1 (1950), 124–127.

[51] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1075–1092. doi:10.1145/3627703.3629578

[52] Foteini Strati, Sara McAllister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. 2024. DéjàVu: KV-cache streaming for fast, fault-tolerant generative LLM serving. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1902, 27 pages.

[53] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: dynamic scheduling for large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 10, 19 pages.

[54] Amos Tversky and Daniel Kahneman. 1974. Judgment under uncertainty: Heuristics and biases. *Science* 185, 4157 (1974), 1124–1131.

[55] Abhishek Vijaya Kumar, Gianni Antichi, and Rachee Singh. 2025. Aqua: Network-Accelerated Memory Offloading for LLMs in Scale-Up GPU Domains. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*.

[56] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. 2024. Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 195–210. doi:10.1145/3694715.3695975

[57] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2025. BurstGPT: A Real-World Workload Dataset to Optimize LLM Serving Systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2* (Toronto ON, Canada) *(KDD '25)*. Association for Computing Machinery, New York, NY, USA, 5831–5841. doi:10.1145/3711896.3737413

[58] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 640–654. doi:10.1145/3694715.3695948

[59] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2026. FastServe: Iteration-Level Preemptive Scheduling for Large Language Model Inference. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[60] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. dLoRA: dynamically orchestrating requests and adapters for LoRA LLM serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 49, 17 pages.

[61] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. 2025. Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 1030–1045. doi:10.1145/3731569.3764815

[62] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: dynamic scaling on GPU clusters for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 30, 16 pages.

[63] Jiale Xu, Rui Zhang, Cong Guo, Weiming Hu, Zihan Liu, Feiyang Wu, Yu Feng, Shixuan Sun, Changxu Shao, Yuhong Guo, Junping Zhao, Ke Zhang, Minyi Guo, and Jingwen Leng. 2024. vTensor: Flexible Virtual Tensor Management for Efficient LLM Serving. *ArXiv* abs/2407.15309 (2024). https://api.semanticscholar.org/CorpusID:271328493

[64] An Yang, Anfeng Li, Baosong Yang, et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[65] An Yang, Baosong Yang, Binyuan Hui, et al. 2024. Qwen2 Technical Report. arXiv:2407.10671 [cs.CL] https://arxiv.org/abs/2407.10671

[66] An Yang, Baosong Yang, Beichen Zhang, et al. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[67] Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. 2025. DeltaZip: Efficient Serving of Multiple Full-Model-Tuned LLMs. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 110–127. doi:10.1145/3689031.3717468

[68] Zahra Yousefijamarani, Xinglu Wang, Qian Wang, Morgan Lindsay Heisler, Taha Shabani, Niloofar Gholipour, Parham Yassini, Hong Chang, Kan Chen, Qiantao Zhang, Xiaolong Bai, Jiannan Wang, Ying Xiong, Yong Zhang, and Zhenan Fan. 2026. HFX: Joint Design of Algorithms and Systems for Multi-SLO Serving and Fast Scaling. *arXiv preprint arXiv:2508.15919* (2026).

[69] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[70] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, Haoran Yang, and Yu Ding. 2025. Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference. In *Proceedings of the 2025 USENIX Annual Technical Conference (ATC)*.

[71] Shan Yu, Yifan Qiao, Mingyuan Ma, Yangmin Li, Shuo Yang, Xinyuan Tong, Yang Wang, Zhiqiang Xie, Yuwei An, Shiyi Cao, Ke Bao, Deepak Vij, Xiaoning Ding, Yichen Wang, Qingda Lu, Zhong Wang, Gao Gao, Harry Xu, Junyi Shu, Jiarong Xing, and Ying Sheng. 2026. Prism: Cost-Efficient Multi-LLM Serving via GPU Memory Ballooning. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[72] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. 2025. BLITZSCALE: fast and live large model autoscaling with O(1) host caching. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation* (Boston, MA, USA) *(OSDI '25)*. USENIX Association, USA, Article 16, 19 pages.

[73] Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. 2025. PQCache: Product Quantization-based KVCache for Long Context LLM Inference. *Proc. ACM Manag. Data* 3, 3, Article 201 (June 2025), 30 pages. doi:10.1145/3725338

[74] Wei Zhang, Zhiyu Wu, Yi Mu, Rui Ning, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. 2026. JITServe: SLO-aware LLM Serving with Imprecise Request Information. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI*

'26). USENIX Association, Santa Clara, CA, USA.

[75] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1506, 50 pages.

[76] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P Xing, et al. 2023. Lmsys-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. *arXiv preprint arXiv:2309.11998* (2023).

[77] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2000, 27 pages.

[78] 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 Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 11, 18 pages.

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