

Model-as-a-Service in the Wild: Characterizing Production LLM Inference Clusters at Scale

Yiyun Zheng
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
yiyunzheng@link.cuhk.edu.cn

Lingyun Yang
Alibaba Group
Hangzhou, China
Shanghai Jiao Tong University
Shanghai, China
yanglingyun.yly@alibaba-inc.com

Yinghao Yu
Alibaba Group
Hangzhou, China
yinghao.yyh@alibaba-inc.com

Guodong Yang
Alibaba Group
Hangzhou, China
luren.ygd@taobao.com

Liping Zhang
Alibaba Group
Hangzhou, China
liping.z@alibaba-inc.com

Minchen Yu*
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
yuminchen@cuhk.edu.cn

Abstract

Large language model (LLM) inference has been widely deployed in modern cloud platforms as Model-as-a-Service (MaaS), serving both online and offline applications on shared, heterogeneous GPU clusters. Recent characterization studies have examined request-level workload patterns, but cluster-wide resource patterns and system-level deployment behaviors remain largely unexplored. This missing view is critical for diagnosing production bottlenecks and for understanding whether and how existing inference techniques should be applied in real MaaS systems.

In this paper, we analyze a six-month production LLM inference trace from xMaaS, a large-scale MaaS platform with over 100K GPUs. xMaaS provides a detailed system view of cluster resources and their allocation across 300K applications and 70K model variants, enabling a *resource-centric* analysis across cluster, application, and model levels. Our study characterizes how heterogeneous GPU resources are supplied and utilized, how application resource demand is distributed and evolves over time, and how model variants and serving modes affect resource allocation on heterogeneous hardware. Based on these observations, we further analyze deployment implications and identify open problems for production MaaS systems, which we hope will motivate future research on resource-efficient LLM inference.

*Corresponding author.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857044

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **General and reference** → **Measurement**.

Keywords: LLM inference, Model-as-a-Service, workload characterization, production traces, heterogeneous GPU clusters, resource management

ACM Reference Format:
Yiyun Zheng, Lingyun Yang, Yinghao Yu, Guodong Yang, Liping Zhang, and Minchen Yu. 2026. Model-as-a-Service in the Wild: Characterizing Production LLM Inference Clusters at Scale. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3828523.3857044

1 Introduction

In recent years, large language models (LLMs) have become the foundation of a wide range of generative AI applications, including chatbots, coding assistants, retrieval-augmented generation, and agent systems. Serving these applications is becoming a critical workload in modern cloud platforms. To meet the growing demand, production clusters continuously run LLM inference as cloud services, often in the form of *Model-as-a-Service (MaaS)*. These services support both user-facing *online* applications with latency and availability requirements, and throughput-oriented *offline* applications such as batch generation [10, 36, 37, 43].

There have been a wide range of system research to optimize LLM inference, such as Key-Value (KV) cache management [8, 14, 19], request scheduling [1, 38], prefill–decode (PD) disaggregation [24, 27, 44], and model scaling [39, 41]. While these techniques bring performance gains in inference latency, throughput, and resource efficiency, their effectiveness largely depends on deployment conditions such as live KV cache demand, resident model variants, resource types, and workload patterns [10, 20].

However, production-scale LLM inference deployments remain poorly characterized. Existing characterizations mainly focus on general machine learning platforms or LLM training workloads [11, 34], whose requirements and behaviors differ from those of LLM inference services. Recent trace studies characterize *request-level* patterns of LLM services, including request arrivals, burstiness, and token lengths [32, 36]. These traces are valuable for workload modeling, but they do not expose *system-level* deployment information, such as resource-usage patterns and cluster-wide views of LLM inference deployments. This lack of visibility creates two practical limitations. First, it hinders the diagnosis of production bottlenecks. Without deployment-level visibility, platform operators cannot pinpoint which system layer is responsible for resource inefficiency or performance degradation. Second, it obscures whether and how existing serving optimizations should be applied in practice, since their effectiveness often depends on deployment-specific conditions.

To bridge this gap, we study a six-month production LLM inference trace from xMaaS, an anonymized production MaaS platform at Alibaba with over 100K GPUs. xMaaS provides a system-level view of cluster resources and their allocation across applications and models, which complements existing request-level LLM inference trace analyses [32, 36]. Building on this view, we conduct a *resource-centric* study that focuses on how GPU resources are supplied, allocated, occupied, and utilized in production. This methodology allows us to characterize production inference across three levels: cluster, application, and model. Specifically, we study the following questions:

- *Cluster level:* How heterogeneous is the resource supply across LLM inference clusters, and how are these resources utilized over time?
- *Application level:* How is resource usage distributed across applications in shared inference infrastructure, and how does the resource demand of individual applications change?
- *Model level:* How do model variants, serving modes, and model architectures affect resource allocation across heterogeneous hardware?

Cluster-level characterization. We begin by characterizing cluster-level resource supply and utilization patterns. We quantify allocated GPU resources with *GPU Time*, measured in GPU-hours, and analyze how GPU Time is distributed across clusters and GPU hardware types and how online and offline workloads share the fleet (§4.1). We then study realized resource utilization across GPU types (§4.2) and examine temporal resource dynamics across different resource categories (§4.3). Our analysis shows that: (i) resource supply is highly heterogeneous across clusters, with the top ten clusters accounting for approximately 60% and the top ten GPU types covering close to 90% of GPU Time, yet the online/offline workload mix varies substantially across GPU types; (ii) resource utilization varies across GPU types, clusters, and workloads; and (iii) compute demand fluctuates over time, while GPU memory utilization typically remains between 80% and 95%.

Application-level characterization. We next analyze applications as deployment units of LLM inference jobs in shared inference infrastructure. We first explore application-level GPU Time distribution and placement across clusters and GPU types (§5.1). We then use representative applications to study intra-application dynamics and their resource demands over time (§5.2). We also examine how KV cache affects GPU memory footprint for both PD-collocated and -disaggregated applications (§5.3). Our analysis reveals that: (i) GPU Time is concentrated among a small fraction of applications: the top 10% account for 79.3% of online and 95.1% of offline GPU Time, while online and offline applications differ substantially in deployment behaviors; (ii) demand fluctuates strongly while GPU allocation barely tracks it, and no GPU type is consistently best across applications; (iii) GPU memory is often heavily reserved but underutilized, with low live KV cache occupancy; and (iv) PD-disaggregated applications expose substantial imbalance for phase-level KV cache occupancy.

Model-level characterization. At the model level, we first compare model resource profiles across online and offline workloads (§6.1). We then analyze model-variant distribution and GPU Time skew (§6.2), and study hot and cold variants to understand their resource contribution and residence behavior (§6.3). We further characterize PD-disaggregated deployments (§6.4) and compare dense models with Mixture-of-Experts (MoE) models (§6.5). Our analysis shows that: (i) online and offline workloads use overlapping hardware and base models, but differ substantially in Streaming Multiprocessor (SM) utilization, with 60% of online GPU Time below 15%; (ii) model variants are numerous and fine-grained, yet the top 10% of variants account for 84.3% of online and 97.3% of offline GPU Time; (iii) cold online variants do not imply short residence, as they can retain GPU memory while contributing little compute activity; (iv) PD-disaggregated deployments mainly appear on large models and high-end GPUs, and exhibit diverse prefill/decode (P/D) sizing ratios; and (v) MoE models introduce strong memory-capacity and multi-GPU placement demand.

Deployment implications and open problems. Based on these observations, we further discuss deployment implications for resource-efficient LLM inference and identify open problems for production-scale clusters. We focus on three key problems. First, production deployments show *severe GPU memory overprovisioning* (§7.1): GPU memory remains highly reserved but often underutilized. This pattern comes from peak-provisioned KV cache pools, memory-resident cold models, and conservative scale-down policies; it also contributes to low compute utilization. It calls for finer-grained memory management for both KV cache and models

and elastic autoscaling mechanisms [35, 40, 41]. Second, *heterogeneous resource management is increasingly important but remains challenging* (§7.2). The best resource configuration depends jointly on workload dynamics, performance requirements, and model characteristics; even within a single application, there is no one-size-fits-all policy [7, 21–23]. We also find sparse model–hardware deployment patterns, suggesting that better runtime support (e.g., kernel autotuning [30, 33]) could expand the feasible configuration space and improve efficiency. Third, *PD disaggregation is not universally applicable in production* (§7.3). Its benefit depends on model scale, workload patterns, and available hardware types [13, 31]. Given the diverse P/D ratios across models and workloads, future systems need elastic P/D resizing that adjusts resources without interrupting inference services.

We believe the observations from xMaaS extend beyond this deployment. They reveal recurring resource patterns in production-scale MaaS systems across cluster, application, and model levels, including skewed GPU consumption, persistent GPU memory occupancy, and heterogeneous model–hardware placement. We have publicly released a sanitized version of the xMaaS trace[1]. Together with the open problems from our production experience, this release supports follow-up research on resource-efficient MaaS systems.

2 Background

2.1 Model-as-a-Service

LLM inference basics. Model-as-a-Service (MaaS) provides hosted LLM inference through cloud services, where users submit requests to deployed model endpoints without managing the underlying serving stack. At runtime, LLM inference consists of two phases: prefill and decode. The prefill phase processes the input prompt in parallel and constructs the initial KV cache; it is generally compute-intensive. The decode phase generates output tokens autoregressively and repeatedly reads the accumulated KV cache; it is often constrained by memory bandwidth, while KV cache capacity limits batching and concurrency [14, 24, 44].

MaaS workloads. Production MaaS runs diverse LLM workloads on shared GPU fleets. A MaaS platform hosts a large number of LLM applications, each backed by a base model or a deployable model variant produced through post-training or service-specific customization. These applications and model variants can differ substantially in popularity, residence time, and resource footprint. The deployed models also span different families, sizes, and architectures, including dense and Mixture-of-Experts (MoE) models. MaaS workloads also differ in service objectives. Online workloads include user-facing services such as chatbots, coding assistants, Retrieval-Augmented Generation (RAG) services, and agent systems, which directly affect user-perceived experience and therefore require low latency and high availability under dynamic request traffic. Offline workloads contain throughput-oriented jobs such as batch generation and evaluation; they have more relaxed latency requirements but often demand sustained GPU capacity.

[1]GitHub: https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2026-maas; Zenodo archive: https://zenodo.org/records/23335733.

Table 1. Comparison with related characterization studies at cluster, application, and model granularity. Notations: ✓ (primary analysis), △ (partial or indirect), ✗ (not covered), – (not reported).

	MLaaS [34]	Acme [11]	ServeGen [36]	**Ours**
Year	2022	2024	2026	2026
#GPUs	6,742	4,704	O(10K)	101,499
#GPU types	8	1	–	30
Duration	2 months	6 months	4 months	6 months
Job types	General ML	LLM train.	LLM infer.	LLM infer.
Cluster-level	✓	✓	✗	✓
Application-level	✗	✗	✗	✓
Model-level	✗	✗	△	✓

2.2 Workload Characterization

Workload characterization is critical for understanding and improving real-world MaaS systems. It helps operators understand how resources are used and where performance bottlenecks arise across clusters and applications. It also guides system designers in deciding whether and how inference optimizations should be applied in production.

Table 1 summarizes representative characterization studies and xMaaS. Existing studies mainly cover general machine-learning clusters [34], LLM training clusters [11], and request-level LLM inference workloads, such as request arrivals, burstiness, and token lengths [32, 36]. They leave the deployment side of production inference largely unexplored, such as cluster-wide resource usage across GPU types and application-level resource demand distribution. This gap becomes more important as modern LLM inference clusters include increasingly diverse GPU types and hardware generations [21, 23]. In contrast, xMaaS provides a *resource-centric* view of production MaaS deployments. We study six months of LLM inference traces from xMaaS, covering over 100K GPUs. The trace exposes GPU resource patterns across clusters, applications, and models, allowing us to analyze how resources are supplied, allocated, occupied, and utilized in production. This view complements request-level traces by revealing the deployment-side resource behavior of production LLM inference services.

3 xMaaS Platform and Trace

3.1 Platform Overview

Fig. 1 shows the main components of xMaaS. Users submit online requests or offline tasks to applications. Each application serves a model variant through one or more instances

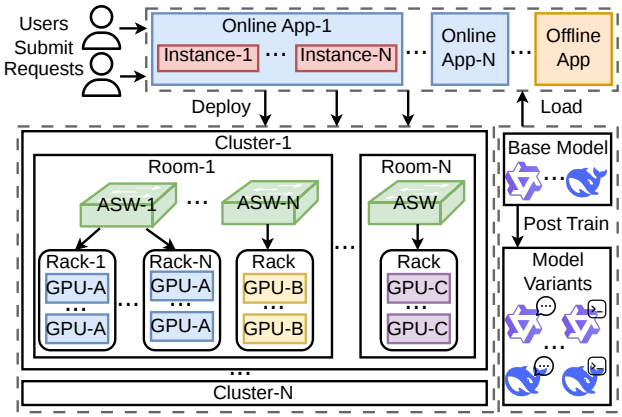


Figure 1. xMaaS overview.

deployed on GPU clusters. We next describe the resource organization, deployment units, and serving configurations.

Cluster resources and heterogeneity. xMaaS spans multiple clusters distributed across different geographic regions. Each cluster contains several server rooms, and each room contains multiple access switch (ASW) domains. An ASW connects a group of racks, and each rack hosts GPU servers. In xMaaS, all racks under the same ASW host the same GPU type, whereas different ASW domains within a room may contain different GPU types. Therefore, GPU resources are naturally heterogeneous at the room and cluster granularity.

Applications and instances. An application is the deployment unit of LLM inference workloads in xMaaS, covering both online and offline services. An application may run one or more instances, each of which is a scheduling and execution unit. An instance retains a fixed GPU allocation throughout its lifetime. An application's resource allocation therefore depends on its instances, their lifetimes, and the GPUs assigned to them.

Base models and model variants. A base model denotes a specific trained model (e.g., DeepSeek-V4), and a model variant denotes a deployable version produced from a base model through post-training or service-specific customization [2, 9, 29]. An instance serves one model variant at a time, while the same model variant can also be deployed on different applications.

Serving engines and modes. Applications use either our in-house serving engine or open-source engines such as SGLang. These engines typically pre-allocate KV cache pools at startup. xMaaS supports both PD-collocated and PD-disaggregated serving. The former executes prefill and decode in the same deployment, while the latter uses separate serving components and transfers KV state between them.

Deployment and scaling policies. Users select the model, GPU type, instance count, and serving configuration for each application. xMaaS supports scheduled scaling that releases resources at predefined times and threshold-based scaling that adds instances according to SM utilization. Online services reserve capacity for traffic bursts and maintain placements on suitable GPU types. Offline services use more aggressive batching and have greater flexibility to use available capacity across clusters and GPU types.

These configurations shape the resource behavior captured by the trace. KV cache preallocation keeps memory reserved, while online headroom and scaling policies affect SM utilization and allocation changes. Although the trace comes from one production MaaS platform, its hardware and workload diversity allows us to examine these behaviors across different deployment settings. We expect qualitative patterns such as resource skew and persistent memory reservation to be relevant to other MaaS systems, while their magnitudes depend on serving-engine configurations, workload mix, and operational policies.

Table 2. Trace overview. GPU Time measured in GPU-hours.

Workloads	Apps.	Insts.	Base models	Model variants	GPU Time
Online	9.7K	1.7M	1,158	10.8K	148.0M
Offline	288.7K	132.6M	638	59.8K	217.9M
Total	298.1K†	134.3M	1,324†	70.1K†	365.8M

†Applications, base models and model variants can overlap across online and offline workloads; the total reports the deduplicated count.

3.2 Trace Overview

The trace covers six months of LLM inference workloads running on xMaaS across 16 regions, 56 clusters, 101,499 GPUs, and 30 GPU types. Table 2 summarizes the scale of online and offline workloads. They account for approximately 40.5% and 59.5% of total GPU Time, respectively.

Metadata collection. We collect deployment metadata as daily instance segments. Each record describes a portion of an instance's observed execution within one calendar day, together with its workload, model, placement, and allocation metadata. Specifically, it includes workload class, application identifier, base model, model variant, model size, model architecture, PD serving mode, cluster and topology placement, GPU type, GPU count, and observed lifetime. We define *GPU Time* as the allocated GPU count multiplied by the observed lifetime in hours within that day. For instances spanning multiple days, we sum segment GPU Time and observed duration across records belonging to the same instance. Instance counts and duration distributions are computed over deduplicated instances.

Performance measurements. The trace includes two types of runtime measurements. First, hardware telemetry is collected per instance at 2-second granularity, including SM utilization, GPU memory utilization, CPU usage, and host memory usage. GPU memory utilization is the ratio of occupied GPU memory to the total memory capacity of the allocated GPUs. Second, the serving engine reports 60-minute application-level aggregates for both online and offline workloads, including Queries Per Second (QPS), average

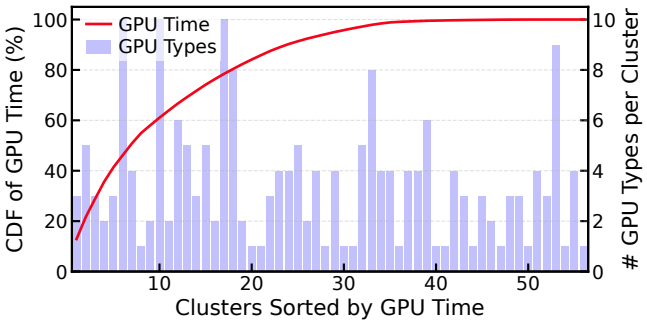


Figure 2. GPU Time distribution and hardware diversity across clusters. Clusters are sorted by GPU Time; the line shows cumulative GPU Time, and bars show distinct GPU types per cluster.

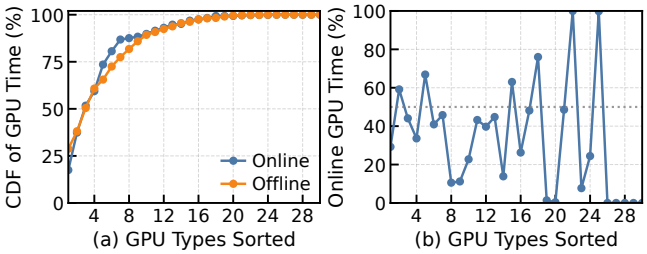


Figure 3. GPU Time distribution and online share of GPU Time across GPU types. GPU types are sorted by GPU Time. (a) Cumulative online and offline GPU Time. (b) Online share of GPU Time per GPU type.

input and output token lengths, average time to first token (TTFT), average time per output token (TPOT), and KV cache occupancy (i.e., the KV-pool used-to-allocated ratio). For PD-disaggregated deployments, KV cache transfer latency is also reported. We use these measurements to analyze resource usage patterns and application-level inference behaviors.

4 Cluster-Level Characterization

4.1 Overall Resource Allocation and Heterogeneity

GPU Time skew and hardware diversity across clusters. Fig. 2 shows that GPU Time is highly skewed across clusters. The largest cluster contributes approximately 10% of total GPU Time, the top ten clusters account for roughly 60%, and the top half of clusters cover more than 90%. Only 14 of the 56 clusters contain a single GPU type; most clusters contain multiple GPU types, and several clusters contain 8–10 GPU types. Taken together, GPU Time is concentrated in a small fraction of clusters, while the hardware within each cluster remains highly heterogeneous.

GPU Time skew and workload mix across GPU types. Fig. 3(a) shows that GPU Time is also highly skewed across the 30 GPU types. The top ten GPU types account for close to 90% of GPU Time for both online and offline workloads. The bottom half of GPU types contributes only 3.5% of GPU Time, which still amounts to 12.9M GPU-hours. Fig. 3(b) further shows that the online share of GPU Time varies

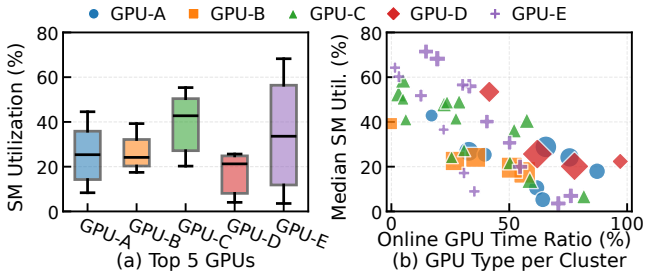


Figure 4. SM utilization of the top five GPU types. (a) GPU-Time-weighted distribution. (b) Each point is one GPU type in one cluster; point size shows GPU Time. X-axis shows online GPU Time ratio. Y-axis shows median SM utilization.

widely across GPU types: some types are nearly fully online, others nearly fully offline, and many sit in between.

> **Observation 1:** GPU Time is highly skewed across clusters and GPU types; clusters are heterogeneous in hardware and GPU types vary in online-offline workload mix.

4.2 SM Utilization across GPU Types and Workloads

Variation in SM utilization across GPU types. Fig. 4(a) compares SM utilization across the top five GPU types sorted by GPU Time. Median SM utilization ranges from about 20% to over 40%, and the interquartile ranges also differ substantially. We next examine how these distributions vary across the clusters and workloads.

Lower SM utilization in online-heavy placements. Fig. 4(b) plots one point per placement of a GPU type in a cluster. Clusters deploying the same GPU type can exhibit markedly different median SM utilization. Across these placements, a higher online share of GPU Time is generally associated with lower utilization. This pattern is consistent with the deployment policies in §3.1: online services reserve headroom for latency and availability, while offline services use more aggressive batching to improve throughput.

> **Observation 2:** SM utilization varies across GPU types, clusters, and workloads, with online-heavy placements generally showing lower utilization.

4.3 Temporal Resource Dynamics

Temporal variation in SM utilization. Fig. 5 compares fleet-wide resource dynamics over a selected week and over the full six-month trace. We select the shaded week because it spans a complete Monday-to-Sunday cycle and is short enough to expose intraday dynamics. During this week, online resource usage follows a clear diurnal pattern across SM utilization, GPU memory, CPU, and host memory. SM utilization for online workloads fluctuates between troughs of 10–15% and peaks of 25–30%, while offline SM utilization is higher, typically between 40–60%, and exhibits less regular

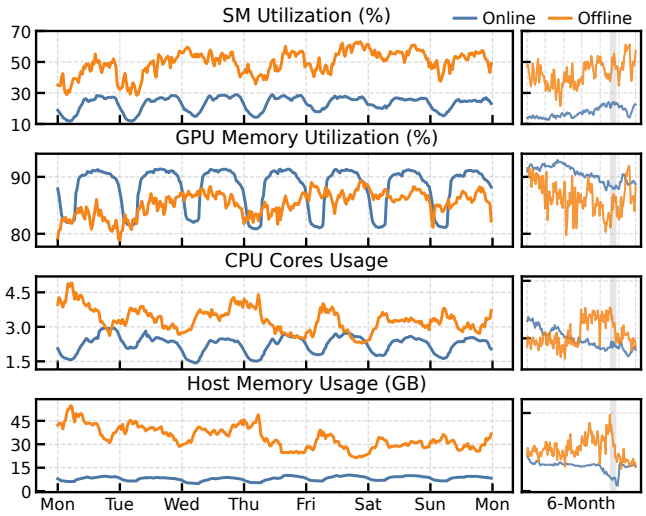


Figure 5. Fleet-wide resource dynamics. Left: selected week. Right: six-month overview with the selected week shaded. Rows show SM utilization, GPU memory utilization, CPU cores usage, and host memory usage.

periodicity. The six-month overview shows the same contrast: online SM utilization remains mostly between 10–25%, while offline is higher and more volatile.

Persistent GPU memory occupancy. In contrast to compute, GPU memory utilization remains high for both workload classes. Online GPU memory utilization typically stays between 80% and 95%, with regular troughs during low-traffic periods. This high baseline is consistent with resident model weights, runtime buffers, and pre-allocated KV cache pools [8, 14]. Offline GPU memory utilization also remains high, between 80% and 90%, but varies more over time. Host-side resources are far less contended: during the selected week, online instances typically use about 1.5–3 CPU cores and 5–10 GB host memory, while offline instances use about 2.5–5 CPU cores and 25–50 GB host memory.

Observation 3: Compute demand fluctuates over time, while GPU memory remains persistently occupied across both online and offline workloads.

5 Application-Level Characterization

5.1 Resource Allocation across Applications

GPU Time skew and deployment scale across applications. Fig. 6(a) shows that GPU Time is concentrated among a small fraction of applications: the top 10% of applications account for 79.3% of online GPU Time and 95.1% of offline GPU Time. Fig. 6(b) shows that online and offline applications also differ in deployment scale. About 70% of online GPU Time comes from applications with at most 100 instances, whereas reaching the same share of offline GPU Time takes applications with up to about 10,000 instances. Offline deployments also span a much wider range: the largest

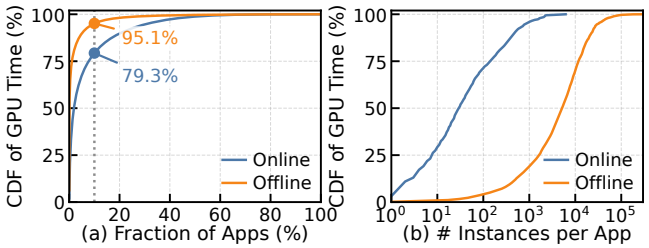


Figure 6. Application GPU Time distribution and deployment scale. CDFs are weighted by GPU Time. (a) Cumulative GPU Time after ranking applications by GPU Time. (b) Number of instances per application-day.

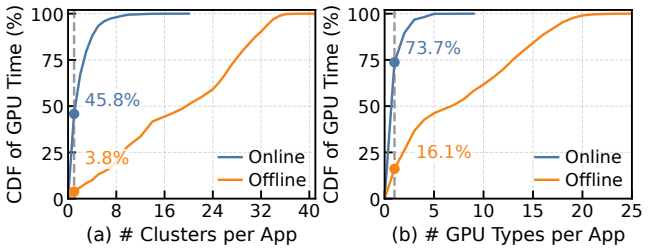


Figure 7. Daily application placement footprint over the six-month trace. CDFs are weighted by GPU Time. (a) Clusters spanned by each application's daily deployment. (b) GPU types used by each application's daily deployment.

online applications remain below 10,000 instances, while offline applications extend to about 300,000 instances.

Concentrated online and dispersed offline placement. We measure placement per application-day to separate daily deployment footprint from six-month cumulative reach. Online applications are relatively concentrated: Fig. 7(a) shows that 45.8% of online GPU Time comes from application-day records in a single cluster. Fig. 7(b) shows that 73.7% comes from records using a single GPU type. Offline applications are much more dispersed. Only 3.8% of offline GPU Time comes from records in one cluster, and only 16.1% comes from records using one GPU type. The offline footprint extends to about 40 clusters and 25 GPU types, far broader than the online footprint. This broader footprint is consistent with the greater placement flexibility of offline workloads.

Observation 4: GPU Time is concentrated among a small fraction of applications; online and offline applications differ in deployment footprint: online deployments are compact, while offline deployments span far more instances, clusters, and GPU types.

5.2 Application-Level Workload Dynamics

Fig. 6 and Fig. 7 characterize resource allocation across all applications over the six-month trace. We next examine individual applications to relate these aggregate patterns to traffic, GPU allocation, and serving latency. Fig. 8 shows two online applications during the selected week in Fig. 5.

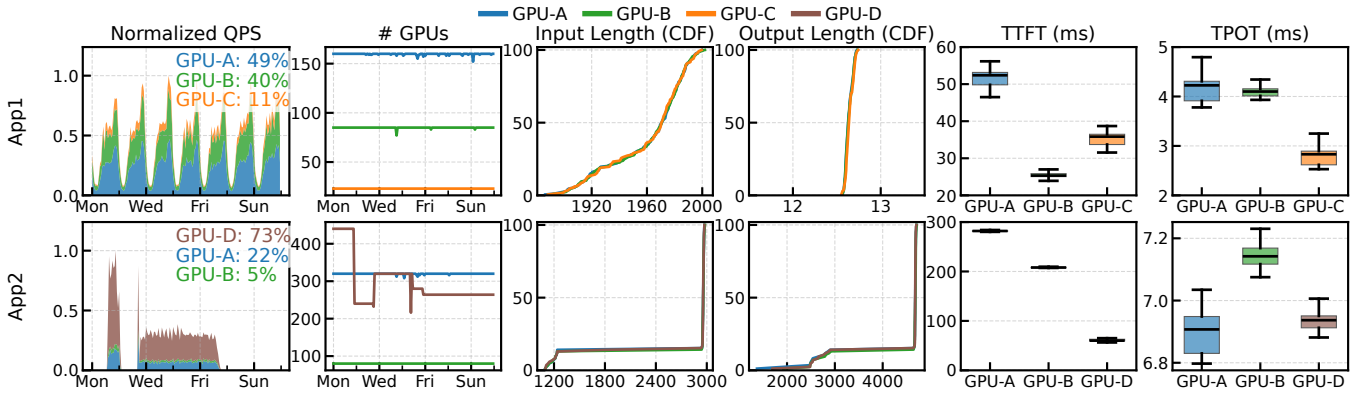


Figure 8. Two online applications during the selected week in Fig. 5. Rows are applications; columns show normalized QPS, allocated GPU cards, CDFs of hourly mean input/output token lengths, TTFT, and TPOT. Serving-engine metrics are hourly aggregates; the CDFs and box plots summarize hourly mean values collected over the selected week highlighted in Fig. 5.

Application 2 has the highest GPU Time among online applications using at least three GPU types, while Application 1 shares GPU-A and GPU-B with it. This enables within- and cross-application comparisons on common hardware. QPS is normalized to conceal product-specific traffic volumes.

Weak coupling between traffic and GPU allocation. Application 1 shows clear daily QPS cycles across GPU types, but its GPU allocation remains nearly flat. Application 2 shows an even stronger decoupling: its QPS remains low for several days, while its GPU allocation changes little. The changes that do occur are concentrated on GPU-D and do not closely track the corresponding QPS series. This indicates conservative capacity adjustment during the period.

Similar average token lengths across GPU types. Within each application, 60-minute average input and output token lengths are similar across GPU types. Application 1's token lengths are tightly concentrated and overlap substantially across GPU types, and Application 2 shows similarly narrow ranges for most input and output lengths. Similar hourly average token lengths do not by themselves explain the observed latency differences.

Latency rankings by application and metric. Fig. 8 shows that the GPU type with the lowest TTFT does not necessarily provide the lowest TPOT. In Application 1, GPU-B has the lowest TTFT (25ms) while GPU-C has the lowest TPOT (2.8ms); in Application 2, GPU-D has the lowest TTFT (65ms) while GPU-A has the lowest TPOT (6.9ms). Among GPU-A and GPU-B, which are shared by both applications, GPU-B has lower TTFT in both cases, with the gap narrowing from about twofold in Application 1 to 1.4 × in Application 2. Their TPOT ordering reverses: GPU-B has lower TPOT in Application 1, whereas GPU-A has lower TPOT in Application 2, although the differences are only a few percent.

Observation 5: Application traffic varies strongly over time, yet GPU allocation barely tracks it; no single GPU type consistently performs best across applications.

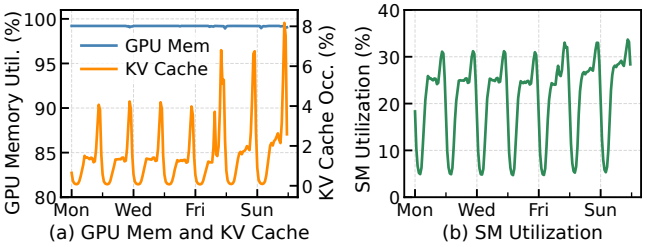


Figure 9. An online application with high GPU Time serving a 4B model during the selected week highlighted in Fig. 5. (a) GPU memory utilization and KV cache occupancy (dual y-axes). (b) SM utilization.

5.3 GPU Memory and KV Cache

High memory reservation with low KV cache occupancy. To examine how physical GPU memory occupancy relates to live KV-cache demand, Fig. 9 shows an online application with high GPU Time that serves a 4B model on about 500 GPUs, each with 96 GB of memory, during the selected week. GPU memory utilization stays near 99%, while KV cache occupancy remains in the low single digits for most of the week (Fig. 9(a)). At the same time, SM utilization still exhibits clear diurnal cycles and peaks above 30% (Fig. 9(b)). This pattern suggests a mismatch between model footprint and GPU memory capacity: a 4B model occupies only part of the 96 GB of GPU memory, so the runtime pre-allocates a large KV cache pool that stays mostly unused even while physical GPU memory remains fully occupied. The high GPU memory occupancy is consistent with the fleet-wide patterns in Fig. 5. The application-level KV cache measurements further reveal that high GPU memory occupancy can coexist with low utilization of the reserved KV pool, a distinction that physical GPU memory telemetry alone does not expose.

Observation 6: Raw GPU memory utilization can overstate live serving demand: GPU memory can be highly reserved but underutilized, with low KV cache occupancy.

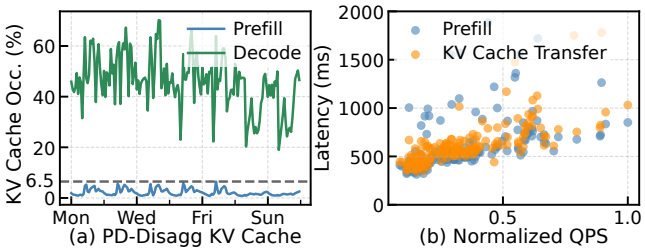


Figure 10. PD-disaggregated online application over the selected week highlighted in Fig. 5. (a) Prefill and decode KV cache occupancy. (b) Prefill latency and KV cache transfer latency versus normalized QPS.

Uneven KV cache occupancy across PD phases. To examine phase-level memory behavior, Fig. 10 follows the online PD-disaggregated application with the highest GPU Time during the same selected week. Separating its prefill and decode measurements reveals how KV-cache occupancy differs between phases, which aggregate GPU memory utilization does not distinguish. Fig. 10(a) shows its prefill and decode KV cache occupancy. Decode occupancy fluctuates between 30% and 60%, whereas prefill occupancy remains low and peaks at about 7%. The prefill pool therefore retains substantial unused KV cache capacity, while the decode pool has higher occupancy.

Overlap between KV transfer and prefill computation. Fig. 10(b) compares prefill latency with the KV cache transfer latency reported by the serving engine. The former includes waiting for and processing local prefill output; the latter covers the decode-side KV transfer stage, including waiting and completion acknowledgment. Both reported metrics increase with normalized QPS and are of comparable magnitude. In this deployment, the engine initiates remote KV transfer before waiting for local prefill output. KV blocks are transferred as each layer completes, allowing communication to overlap with subsequent prefill computation. The full reported KV transfer latency is not an additional delay after prefill; only the remaining wait for transfer completion adds latency at that point. Compute-bound execution or ample network capacity can keep this remaining wait small, whereas bandwidth constraints can increase it.

Observation 7: PD-disaggregated applications show prefill and decode KV cache occupancy differ sharply. KV transfer latency is comparable to prefill latency, while pipelined KV transfer overlaps with prefill execution.

6 Model-Level Characterization

6.1 Model Resource Profiles

Overlapping hardware and model scales with divergent SM utilization. Fig. 11(a)–(c) show that online and offline workloads are not cleanly separated by GPU peak compute capacity, GPU memory capacity, or model size. About 80% of GPU Time for both classes runs on GPUs below 200 TFLOPS

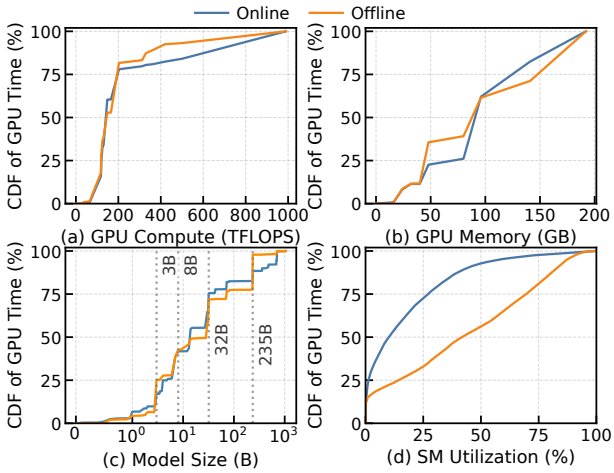


Figure 11. Model-level resource profiles for online and offline workloads. All CDFs are weighted by GPU Time. (a) GPU peak compute capacity. (b) GPU memory capacity. (c) Model size. (d) SM utilization.

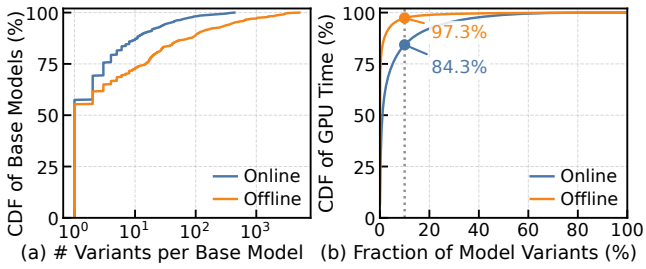


Figure 12. Base-model fan-out and model-variant GPU Time. (a) Unweighted CDF of model variants per base model. (b) Cumulative GPU Time of ranked model variants by GPU Time.

(Fig. 11(a)). Their GPU memory distributions also largely overlap: about 60% of GPU Time in both classes runs on GPUs below 100 GB, and both extend to the largest memory capacities (Fig. 11(b)). Model size shows a similar pattern: models below 8B account for about 40% of GPU Time in both classes, and both reach about 75% of GPU Time by 32B (Fig. 11(c)). The main differences appear in the largest model buckets, where offline workloads show a clear jump at 235B and online workloads have a longer tail beyond 235B.

Despite overlapping hardware and model scales, online and offline workloads differ sharply in SM utilization. Fig. 11(d) shows that online workloads contribute less than 10% of GPU Time above 50% SM utilization, compared with more than 40% for offline workloads. Moreover, 60% of online GPU Time is below 15% SM utilization, compared with 30% for offline workloads.

Observation 8: Online and offline workloads span overlapping ranges of GPU and model size, while online GPU Time is more concentrated at low mean SM utilization.

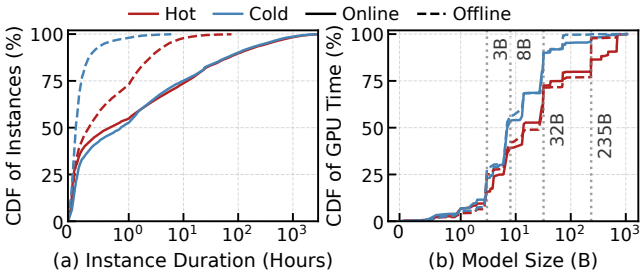


Figure 13. Hot and cold model variants. (a) Unweighted instance lifetime CDF. (b) GPU-Time-weighted model size CDF. Hot variants are the top decile by GPU Time.

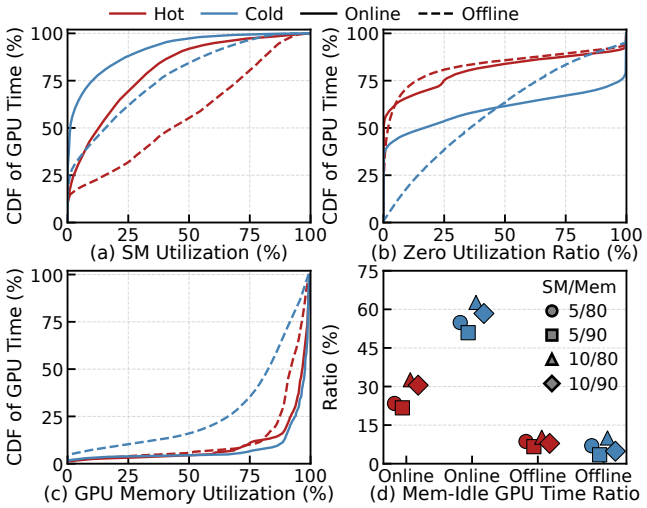


Figure 14. Resource utilization of hot and cold model variants. CDF panels are weighted by GPU Time. (a) SM utilization. (b) Zero-utilization ratio. (c) GPU memory utilization. (d) Memory-idle GPU Time under SM/memory thresholds.

6.2 Model Variants and GPU Time Skew

Model-variant diversity across base models. For online workloads, the trace covers 1,158 base models and 10.8K model variants (9.3 variants per base model), while offline workloads cover fewer base models (638) but expand to 59.8K model variants (93.7 variants per base model). Despite having about 5.5× as many model variants, offline workloads account for only 1.5× as much GPU Time as online workloads (Table 2). Fig. 12(a) shows that this gap persists across the per-base-model distribution. About 55% of base models have only one variant in both workloads. About 85% of online base models have fewer than 10 variants, compared with 75% for offline. At the other extreme, the most variant-rich base models exceed 5K variants. Aggregating by base model would hide this diversity in deployable model variants.

GPU Time skew across model variants. Fig. 12(b) ranks model variants by GPU Time: within each class, the top 10% of variants account for 84.3% and 97.3% of online and offline GPU Time, respectively.

> **Observation 9:** Variant counts vary widely across base models, while the top decile of variants accounts for the vast majority of GPU Time.

6.3 Hot and Cold Model Variants

Within each workload class, we rank model variants by their six-month GPU Time and define the top 10% as *hot* variants and the remaining 90% as *cold* variants. This classification is based solely on GPU Time contribution.

Instance lifetimes of hot and cold variants. Fig. 13(a) shows that the lifetime distributions of instances belonging to hot and cold online variants are nearly identical: about half of instances in both categories last under one hour, while the longest instances extend beyond 1,000 hours. Cold status in online serving therefore does not translate into short lifetimes. In contrast, offline instance lifetimes are shorter overall, and cold offline instances are substantially shorter-lived than hot offline instances: about 95% of cold offline instances last under one hour, compared with about 70% for hot offline instances. The lifetime implications of cold status are therefore workload-dependent: cold online variants can persist as long as hot ones, while cold offline variants are predominantly short-lived.

Smaller model sizes among cold variants. Fig. 13(b) shows that GPU-Time rank is related to, but not determined solely by, model size. Cold variants accumulate GPU Time rapidly in the 3B–8B range, especially for offline workloads. Online cold variants also rise faster than hot ones through the mid-sized range: by 32B, cold variants have accumulated about 90% of their GPU Time, compared with about 75% for hot variants. The distinction appears mainly in the tail: hot variants retain more GPU Time above 32B, offline hot variants show a clear jump at 235B, and online hot variants extend farthest into the largest model sizes.

Lower SM utilization among cold variants. Fig. 14(a) shows that hot variants consistently exhibit higher SM utilization than cold variants. For the same 75% of GPU Time, SM utilization stays below 10% for online cold variants but rises to nearly 30% for online hot variants. Fig. 14(b) further sharpens this contrast: using a 50% zero-utilization threshold, about 40% of cold GPU Time exceeds the threshold in both classes, compared with less than 20% for hot variants. More than 20% of online cold GPU Time has a zero-utilization ratio near 100%, exposing long no-compute intervals rather than merely low average SM utilization.

High memory-idle ratios among cold online variants. We define a *memory-idle* interval as one with low SM utilization and high GPU memory utilization, capturing periods where GPUs retain model weights, runtime buffers, and reserved KV cache capacity while doing little computation. Fig. 14(c) shows that hot variants in both workload classes and online cold variants spend over 80% of GPU Time above 80% memory utilization; cold offline variants are lower overall. Across four SM/memory threshold pairs in Fig. 14(d), cold

Table 3. Scale of online workloads by PD serving mode over the six-month trace.

Mode	Apps.	Insts.	Base models	Model variants	GPU Time
PD-Colloc.	9.4K	1.6M	1148	10.7K	133.6M
PD-Disagg.	477	69.0K	146	384	14.4M

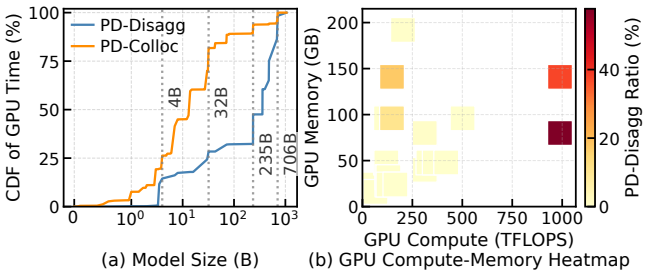


Figure 15. PD serving mode comparison for online workloads. The CDF in (a) is weighted by GPU Time. (a) Model size CDF by PD serving mode. (b) GPU types positioned by peak compute and memory capacity, colored by the PD-disaggregation share of online GPU Time on each GPU type.

online variants have the highest memory-idle ratio, roughly 50–65% of GPU Time. Hot online variants are lower, roughly 20–30%, and both offline groups remain below about 15%. This makes cold online variants a residency problem rather than merely a low-compute tail.

Observation 10: Cold online variants can be long-lived and memory-resident despite contributing little GPU Time and compute activity.

6.4 PD Serving Modes

xMaaS provides two serving modes: PD-collocation and PD-disaggregation. PD-disaggregation separates prefill and decode execution into distinct online serving components, while PD-collocation keeps the two phases in the same deployment [24, 27, 44]. We analyze serving modes for online workloads, where the need for PD-disaggregation arises.

PD disaggregation on large models and high-end GPUs. Table 3 shows that PD-disaggregation is a small minority of online deployments by application, instance, and model-variant counts, but it still consumes 14.4M GPU-hours. Fig. 15(a) shows that the two modes differ strongly in their model-size distributions: for PD-disaggregation, models below 32B account for only about 30% of GPU Time, and more than half of PD-disaggregation GPU Time is consumed by models larger than 235B. In contrast, models above 235B account for less than 10% of PD-collocation GPU Time, indicating that PD-disaggregation is concentrated on larger models. On the hardware side, Fig. 15(b) shows that PD-disaggregation is concentrated on GPU types with higher peak compute capacity and larger memory capacity, with an even stronger association with compute capacity.

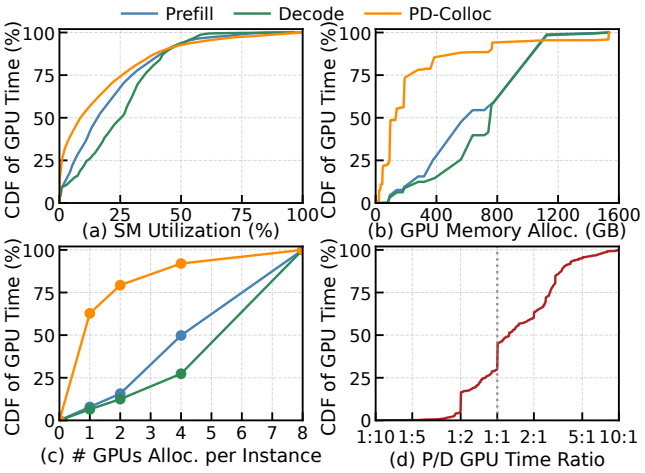


Figure 16. Resource profiles of PD-collocation instances and PD-disaggregated prefill and decode instances. All CDFs are weighted by GPU Time. (a) SM utilization. (b) Per-instance GPU memory allocation. (c) GPUs allocated per instance. (d) Prefill/decode GPU Time ratio for PD-disaggregation.

Prefill and decode resource profiles. Fig. 16(a) shows higher observed SM utilization for decode instances than for prefill instances, with both exceeding PD-collocated instances. Observed utilization depends on deployment-level provisioning as well as phase characteristics. Reserving prefill capacity to absorb prompt bursts and protect TTFT can lower average prefill SM utilization, even though prefill is generally compute-intensive. Fig. 16(b) compares GPU memory allocation per instance. Prefill and decode have broadly similar allocation profiles, with both generally receiving more memory per instance than PD-collocated instances. In xMaaS, PD-disaggregated deployments are confined to a single ASW and use the same GPU type across phases (§3.1). Per-instance memory allocation also depends on the number of GPUs assigned to each instance.

Fig. 16(c) compares allocated GPU count per instance: single-GPU instances account for more than 60% of PD-collocation GPU Time but less than 10% of PD-disaggregation GPU Time, while PD-disaggregation is dominated by four- and eight-GPU instances. Beyond per-instance allocation, Fig. 16(d) shows that the prefill-to-decode GPU Time ratio spans from extreme decode-heavy (1:10) to extreme prefill-heavy (10:1). A balanced 1:1 ratio accounts for about 20% of PD-disaggregation GPU Time, while more than half comes from deployments with a prefill-heavy ratio. This wide spread makes phase balance a per-model, per-application sizing decision rather than a fixed ratio.

Observation 11: PD-disaggregated deployments are concentrated on large models and high-end GPUs, and their P/D balance varies by model and application rather than following a fixed ratio.

Table 4. Scale of dense and MoE workloads (online and offline combined) over the six-month trace.

Architecture	Apps.	Insts.	Base models	Model variants	GPU Time
Dense	260.4K	118.4M	1169	68.8K	270.3M
MoE	38.3K	15.9M	155	1.4K	95.6M

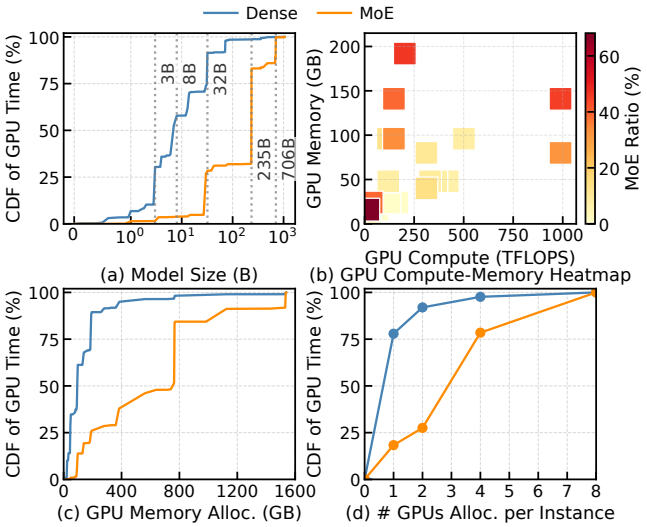


Figure 17. Dense and MoE resource profiles for online and offline workloads combined. CDF panels are weighted by GPU Time. (a) Model size distribution. (b) GPU types positioned by peak compute and memory capacity, colored by the MoE share of GPU Time on each GPU type. (c) Per-instance GPU memory allocation. (d) GPUs allocated per instance.

6.5 Dense and MoE Architectures

High GPU Time per MoE variant. Table 4 compares dense and MoE [4, 16] workloads across the full six-month trace (online and offline combined). MoE workloads have far fewer base models, model variants, and applications than dense workloads, yet still account for about 26% of combined GPU Time. Their share of GPU Time remains stable over the six-month observation window. Per model variant, MoE consumes about 18 × as much GPU Time as dense, consistent with its greater use of multi-GPU deployments.

Large MoE models on high-memory GPUs. Fig. 17(a) shows that MoE models are substantially larger than dense models, with GPU Time concentrated around 30B, 235B, and 700B. The 235B class alone accounts for about half of MoE GPU Time. For dense models, those above 32B contribute less than 10% of GPU Time. In this trace, GPU Time from very large models is therefore dominated by MoE architectures.

Fig. 17(b) shows that MoE models are concentrated on GPUs with large memory capacity. Two GPU types in the lower-left of Fig. 17(b) show disproportionately high MoE fractions; instance metadata indicates that these correspond to dedicated offline testing environments where MoE model evaluation dominated GPU Time.

Multi-GPU deployment of MoE models. Fig. 17(c) shows that for MoE, per-instance memory allocations above 400 GB account for about 60% of GPU Time, with more than 10% exceeding 800 GB. For dense models, per-instance allocations are much smaller: about 90% of dense GPU Time uses 200 GB or less per instance. In Fig. 17(d), single-GPU instances account for about 80% of dense GPU Time but less than 20% of MoE GPU Time. MoE models predominantly use four- or eight-GPU instances, with four-GPU instances accounting for about half of MoE GPU Time and eight-GPU instances for about 20%. Because model size reflects total parameter count rather than active parameters, these figures primarily capture memory-capacity and multi-GPU placement demand.

> **Observation 12:** MoE workloads have fewer model variants but consume more GPU Time per variant and typically use larger GPU memory allocations and more GPUs per instance.

7 Deployment Implications

In this section, we discuss the implications for production LLM inference deployments and highlight open problems exposed by our resource-centric characterization. We focus on three key problems as follows.

7.1 GPU Memory Overprovisioning

Our measurements reveal severe GPU memory overprovisioning in production MaaS: GPU memory is often highly reserved but only partially used by live serving demand. This pattern has three main causes. First, KV cache pools are provisioned for peak load but can remain underutilized at runtime (Fig. 9). Second, many cold online model variants stay resident on GPUs for long periods while contributing little compute activity (Fig. 14). Third, online services use conservative scale-down policies to preserve service quality (Fig. 8). Together, these factors keep GPU memory occupied and also contribute to low compute utilization. We discuss three directions for improving memory efficiency and the production challenges they face.

Dynamic memory pools for KV cache. The first direction is fine-grained memory management for dynamic KV cache demand. While recent systems propose reclaiming or sharing idle GPU memory across collocated services [6, 35, 40], production deployment introduces two constraints. First, such mechanisms must work across diverse GPU types and architecture generations; xMaaS spans 30 GPU types, while many existing mechanisms rely on vendor- or runtime-specific memory APIs. Second, memory sharing weakens fault isolation. Collocating online services in a shared memory or compute domain can couple their failures, which is risky for long-running online instances that can stay active for over 1,000 hours (Fig. 13(a)). Thus, practical KV cache

memory sharing must be portable across heterogeneous GPUs and preserve strong isolation for online services.

Memory sharing across cold model variants. The second direction is to reduce the memory footprint of cold model variants through sharing. For variants that share the same base-model weights, adapter-serving systems provide one way to avoid duplicating those weights [2, 12, 29]. Applying sharing across production deployments also requires compatibility among inference frameworks, memory layouts, communication libraries, and quantization formats. Production MaaS platforms often run multiple serving runtimes, and thus a sharing mechanism that assumes a single framework or uniform runtime covers only part of the deployment. Effective cold-model sharing therefore needs compatibility-aware grouping and cross-runtime support.

Elastic scaling. The third direction is elastic scaling for online deployments. Prior work accelerates model loading and scale-out [5, 10, 41] and adapts resource allocation through hierarchical autoscaling [25]. Our observations highlight a complementary problem: safe scale-down is also critical in production. Online applications can keep stable GPU allocations even when traffic stays low for extended periods, because releasing capacity may remove warm model state and hurt burst tolerance. A production autoscaler must therefore reclaim idle GPU memory and compute capacity without causing performance degradation during sudden traffic recovery. This requires scale-down decisions that are fine-grained, safe, and robust under changing workload patterns.

> **Open Problem 1:** How can production MaaS systems (i) elastically reclaim peak-provisioned but underutilized KV cache memory across diverse GPU types, (ii) share GPU memory across cold model variants with heterogeneous serving runtimes, and (iii) safely scale down idle online capacity without affecting performance?

7.2 Heterogeneous Resource Management

Our characterization shows that heterogeneous resource management is becoming a central problem for production MaaS. At O(100K)-GPU scale, hardware heterogeneity is unavoidable: xMaaS already spans 30 GPU types, and future fleets will include more GPU generations and specialized accelerators. At the same time, MaaS workloads are increasingly diverse. Different applications, agent services, model architectures, and service modes can exhibit distinct performance requirements and resource demands.

Extensible placement policies for heterogeneous resources. Recent systems explore heterogeneous resource configurations for LLM inference [7, 21–23]. However, many of them assume a fixed set of hardware candidates or target specific workloads. Our observations suggest that production MaaS requires a more *scalable and extensible* approach. The best resource configuration depends jointly on workload dynamics, latency requirements, model architecture, serving mode, and resource availability. Even within a single application, the preferred GPU type can change with request patterns and performance targets (Fig. 8), making static GPU rankings or one-size-fits-all placement policies insufficient. Moreover, such an approach should adapt to evolving hardware and workloads as new GPU generations, specialized accelerators, applications, and workloads enter production.

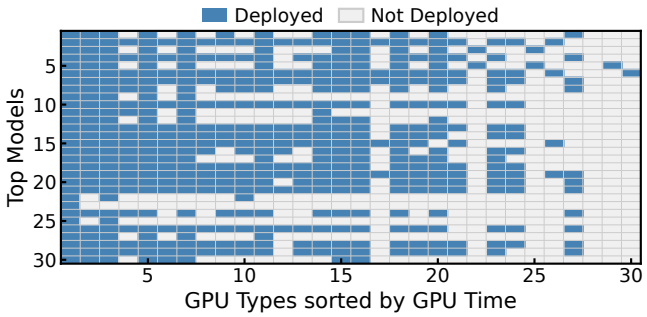


Figure 18. Deployment coverage of the top 30 models across GPU types. Rows are models sorted by GPU Time; columns are GPU types sorted by GPU Time; colored cells indicate observed deployments.

Non-uniform deployment coverage. We also find that model deployment coverage is highly non-uniform across GPU types. Fig. 18 shows that only 53.4% of the top-30-model × GPU-type combinations appear in the trace. The deployment matrix is denser on the left than on the right: GPU types with higher total GPU Time tend to host a broader range of these models, whereas less-used GPU types exhibit more limited observed model coverage. This sparse deployment surface partly reflects incomplete runtime support, such as missing kernels, quantization support, or communication backends on certain GPU types. This suggests a large optimization opportunity: if runtime support expands, many previously unavailable model–hardware placements may become feasible without changing hardware, thereby enlarging the space for resource scheduling and management. Recent work on automatic and agent-based runtime optimization points to this direction, such as CuTe-based CUDA generation, Intel GPU porting, and accelerator-specific kernel tuning [28, 30, 33, 42].

> **Open Problem 2:** How can production MaaS systems build scalable mechanisms for heterogeneous resource management that adapt to diverse workloads, evolving GPU types, and non-uniform runtime support?

7.3 Applicability of PD-Disaggregation

Constraints on PD disaggregation. PD disaggregation is an active direction for improving LLM inference efficiency [13, 24, 31, 44]. For latency-sensitive applications, it can reduce interference and enable independent scaling and resource specialization. Its practical benefit depends on whether these gains outweigh KV cache transfer and

operational costs. In xMaaS, PD disaggregation accounts for only a minority of deployments (Table 3, Fig. 15). Three production factors shape this tradeoff.

First, the heterogeneous-resource benefit of PD disaggregation is limited by *production hardware organization*. PD disaggregation can improve efficiency by placing prefill and decode on different resource types, but high-speed interconnect domains in production are often homogeneous; for example, racks under the same ASW in xMaaS use the same GPU type (Fig. 1). On the *hardware side*, datacenter expansion usually follows GPU generations and deployment batches, and thus local high-speed domains are not built as arbitrary mixtures of accelerators. On the *operational side*, homogeneous local domains simplify compatibility, capacity planning, and support for diverse workloads. As a result, mapping prefill and decode to different accelerator classes often requires crossing local hardware domains, where KV cache transfer can reduce or even erase the benefit of phase specialization. Software-level designs such as prefill-as-a-service partially address this issue by pooling prefill capacity across serving deployments [26], but they still need to handle locality, scheduling, and KV cache transfer overhead.

Second, the benefit of PD disaggregation depends on *workload characteristics and service requirements*. Our trace contains many small- and medium-sized models; models with up to 32B parameters account for about three quarters of online GPU Time (Fig. 11(c)), while PD-disaggregated serving is concentrated on larger models (Fig. 15). This pattern is consistent with selective adoption based on workload needs. When collocated serving already meets latency and throughput requirements, PD disaggregation introduces additional transfer and management costs with limited benefit.

Third, PD disaggregation *increases operational complexity*. Compared with PD-collocated serving, PD-disaggregated serving requires separate resource pools, cross-instance KV cache transfer, coordinated scaling, and more complex monitoring and failure handling. These costs affect observability, reliability, and day-to-day operation in large-scale MaaS platforms. Therefore, production systems apply PD disaggregation selectively, only when its performance or efficiency benefit outweighs its additional management cost.

Elastic P/D resizing. For workloads that benefit from PD disaggregation, a further challenge is adapting prefill and decode capacity as demand changes [15, 18]. The diverse P/D ratios in Fig. 16(d) motivate provisioning tailored to individual workloads. Elastic P/D resizing must handle two requirements. First, resource allocation should account for prefill and decode demand as well as TTFT and TPOT targets. Second, moving requests across P/D pools introduces KV cache transfer costs and locality constraints. To limit the remaining wait for KV transfer completion after prefill (§5.3), resizing decisions should jointly account for prefill compute capacity and available KV transfer bandwidth. These adjustments must preserve service availability and reliability.

Open Problem 3: How can production MaaS systems make PD disaggregation elastic by resizing P/D resources under changing workloads while preserving service availability and reliability?

8 Related Work

A large body of work has studied *resource-efficient LLM inference*. To improve GPU memory efficiency, prior systems optimize KV cache management, reuse, and transfer across storage tiers [8, 14, 19, 20, 27, 37]. Other work improves resource efficiency through elastic scaling and cold-start optimization [5, 10, 25, 39, 41]. To improve GPU utilization, existing systems optimize batching and scheduling [1, 38], PD-disaggregation [13, 15, 18, 24, 26, 31, 44], and heterogeneous resource configurations [7, 21–23]. A complementary line of work exploits LoRA adapters to share base-model weights [2, 9, 12, 29], or coordinates placement and resource sharing across colocated models [3, 6, 17, 35, 40].

Our resource-centric trace analysis complements these systems with deployment-level evidence from production MaaS platforms. It exposes fundamental resource patterns that are difficult to observe from request-level traces or benchmarks, providing empirical grounding for designing future resource-efficient MaaS systems.

9 Conclusion

We characterize six months of production LLM inference trace on xMaaS, a large-scale MaaS platform with over 100K GPUs and 30 GPU types. Through a resource-centric analysis across clusters, applications, and models, we reveal how GPU resources are supplied, allocated, occupied, and utilized in production deployments. Our observations further expose open problems for production MaaS systems to motivate future research, including GPU memory efficiency, heterogeneous resource management, and elastic PD-disaggregation.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their helpful feedback. Lingyun Yang was supported in part by the Joint Postdoctoral Research Workstation established by Alibaba DAMO Academy and Shanghai Jiao Tong University. This work was supported in part by the CUHK-Shenzhen Research Grant (UDF01003466) and the Guangdong Provincial Key Laboratory of Mathematical Foundations for Artificial Intelligence (2023B1212010001).

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*. USENIX Association, Santa Clara, CA, 117–134.

[2] 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 (MLSys '24)*, Vol. 6. MLSys, 1–13.

[3] 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 (ICML '24) (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 11905–11917.

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

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

[6] 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, Boston, MA, 587–595.

[7] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. 2024. Mélange: Cost Efficient Large Language Model Serving by Exploiting GPU Heterogeneity. arXiv:2404.14527

[8] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. MemServe: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool. arXiv:2406.17565

[9] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations (ICLR '22)*.

[10] 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, Boston, MA, 57–72.

[11] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. 2024. Characterization of Large Language Model Development in the Datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)*. USENIX Association, Santa Clara, CA, 709–729.

[12] Nikoleta Iliakopoulou, Jovan Stojkovic, Chloe Alverti, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2025. Chameleon: Adaptive Caching and Scheduling for Many-Adapter LLM Inference Environments. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 217–231. doi:10.1145/3725843.3756083

[13] Yibo Jin, Tao Wang, Huimin Lin, Mingyang Song, Peiyang Li, Yipeng Ma, Yicheng Shan, Zhengfan Yuan, Cailong Li, Yajing Sun, Tiandeng Wu, Xing Chu, Ruizhi Huan, Li Ma, Xiao You, Wenting Zhou, Yunpeng Ye, Wen Liu, Xiangkun Xu, Yongsheng Zhang, Tiantian Dong, Jiawei Zhu, Zhe Wang, Xijian Ju, Jianxun Song, Haoliang Cheng, Xiaojing Li, Jiandong Ding, Hefei Guo, and Zhengyong Zhang. 2024. P/D-Serve: Serving Disaggregated Large Language Model at Scale. arXiv:2408.08147

[14] 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 (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[15] Ruiqi Lai, Hongrui Liu, Chengzhi Lu, Zonghao Liu, Siyu Cao, Siyang Shao, Yixin Zhang, Luo Mai, and Dmitrii Ustiugov. 2025. TokenScale: Timely and Accurate Autoscaling for Disaggregated LLM Serving with Token Velocity. arXiv:2512.03416

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

[17] 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.

[18] Junhan Liao, Minxian Xu, Wanyi Zheng, Yan Wang, Kejiang Ye, Rajkumar Buyya, and Chengzhong Xu. 2026. DOPD: A Dynamic PD-Disaggregation Architecture for Maximizing Goodput in LLM Inference Serving. *IEEE Transactions on Services Computing* 19, 2 (2026), 1134–1147. doi:10.1109/tsc.2026.3670011

[19] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. 2025. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. arXiv:2510.09665

[20] Zedong Liu, Xinyang Ma, Dejun Luo, Hairui Zhao, Bing Lu, Wenjing Huang, Yida Gu, Xingchen Liu, Zheng Wei, Jinyang Liu, Dingwen Tao, and Guangming Tan. 2026. KVServe: Service-Aware KV Cache Compression for Communication-Efficient Disaggregated LLM Serving. arXiv:2605.13734

[21] Yixuan Mei, Zikun Li, Zixuan Chen, Shiqi Pan, Mengdi Wu, Xupeng Miao, Zhihao Jia, and K. V. Rashmi. 2026. Coral: Cost-Efficient Multi-LLM Serving over Heterogeneous Cloud GPUs. arXiv:2605.04357

[22] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2025. Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 586–602. doi:10.1145/3669940.3707215

[23] Zizhao Mo, Jianxiong Liao, Huanle Xu, Zhi Zhou, and ChengZhong Xu. 2025. Hetis: Serving LLMs in Heterogeneous GPU Clusters with Fine-grained and Dynamic Parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 1710–1724. doi:10.1145/3712285.3759784

[24] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA '24)*. IEEE, 118–132. doi:10.1109/isca59077.2024.00019

[25] Archit Patke, Dhemath Reddy, Saurabh Jha, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2025. Hierarchical Autoscaling for Large Language Model Serving with Chiron. arXiv:2501.08090

[26] Ruoyu Qin, Weiran He, Yaoyu Wang, Zheming Li, Xinran Xu, Yongwei Wu, Weimin Zheng, and Mingxing Zhang. 2026. Prefill-as-a-Service: KVCache of Next-Generation Models Could Go Cross-Datacenter. arXiv:2604.15039

[27] 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.

[28] Tara Saba, Zhiyang Chen, Jikai Jason Li, Anne Ouyang, Xujie Si, and Fan Long. 2026. CuTeGen: An LLM-Based Agentic Framework for Generation and Optimization of High-Performance GPU Kernels using CuTe. arXiv:2604.01489

[29] 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 (MLSys '24)*, Vol. 6. MLSys, 296–311.

[30] Marcin Spoczynski, Daniel Fleischer, Moshe Berchansky, Gabriela Ben-Melech Stan, Shira Guskin, Weilin Xu, Adam Siemieniuk, and Alexander Heinecke. 2026. Xe-Forge: Multi-Stage LLM-Powered Kernel Optimization for Intel GPU. arXiv:2605.26118

[31] Chao Wang, Pengfei Zuo, Zhangyu Chen, Yunkai Liang, Zhou Yu, and Ming-Chang Yang. 2025. Prefill-Decode Aggregation or Disaggregation? Unifying Both for Goodput-Optimized LLM Serving. arXiv:2508.01989

[32] 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 (KDD '25)*. Association for Computing Machinery, New York, NY, USA, 5831–5841. doi:10.1145/3711896.3737413

[33] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. 2025. Astra: A Multi-Agent System for GPU Kernel Performance Optimization. arXiv:2509.07506

[34] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*. USENIX Association, Renton, WA, 945–960.

[35] 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 (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 1030–1045. doi:10.1145/3731569.3764815

[36] Yuxing Xiang, Xue Li, Kun Qian, Yan Zhang, Wenyuan Yu, Ennan Zhai, Xin Jin, and Jingren Zhou. 2026. ServeGen: Workload Characterization and Generation of Large Language Model Serving in Production. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '26)*. USENIX Association, Renton, WA, 1845–1859.

[37] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 94–109. doi:10.1145/3689031.3696098

[38] 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.

[39] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Zirui Wang, Yue Cheng, Wei Wang, Ao Wang, and Ruichuan Chen. 2026. FaaScale: Unlocking Fast LLM Scaling for Serverless Inference. In *Proceedings of Machine Learning and Systems (MLSys '26)*, Vol. 8. MLSys, 353–366.

[40] 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 *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI '26)*. USENIX Association, Seattle, WA, 75–93.

[41] 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 *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*. USENIX Association, Boston, MA, 275–293.

[42] Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. 2025. AccelOpt: A Self-Improving LLM Agentic System for AI Accelerator Kernel Optimization. arXiv:2511.15915

[43] 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 *Advances in Neural Information Processing Systems (NeurIPS '24)*, Vol. 37. Curran Associates, Inc., 62557–62583. doi:10.52202/079017-2000

[44] 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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*. USENIX Association, Santa Clara, CA, 193–210.