LLMFabric: Unifying Decentralized HPC Clusters for Heterogeneous LLM Serving *(Operational Systems)*

Xiaozhe Yao*
ETH Zurich

Youhe Jiang*
University of Cambridge

Ilia Badanin
EPFL

Qinghao Hu
MIT

Binhang Yuan
HKUST

Eiko Yoneki
University of Cambridge

Imanol Schlag
ETH Zurich

Ana Klimovic
ETH Zurich

Abstract

Sovereign AI initiatives increasingly rely on federated, heterogeneous GPU clusters, yet these environments are optimized for long-running batch jobs rather than the interactive, multi-tenant workloads of modern LLM serving. LLMFabric is a decentralized serving system that turns existing Slurm-managed HPC clusters into a shared, cross-institutional inference platform. It layers a unified API over diverse serving engines (e.g., vLLM, SGLang) and heterogeneous hardware, using a peer-to-peer gossip network with a CRDT-based registry for service discovery, health monitoring, and fault-tolerant routing. We design a heterogeneity-aware scheduler that jointly decides model placement and parallelism strategies across mixed GPU types using a simulator to estimate execution time and constraint programming to balance latency and resource utilization goals. LLMFabric has been in continuous operation for over 16 months serving more than 13 million requests and 15 billion tokens over 142 models to over 500 researchers across multiple institutions. We open-source LLMFabric and will release anonymized traces to support future research on LLM serving system design.

1 Introduction

In recent years, the capabilities of machine learning models and their range of applications have grown tremendously. The rapid growth of large language models (LLMs) has enabled a wide range of downstream applications, from chatbot services to coding assistants. As the catalog of models and their usage continues to grow, efficient model serving has become a necessary component of modern AI infrastructure. In fact, inference often dominates compute cycles even compared to model training [12, 55].

While commercial AI platforms (e.g., OpenAI, Anthropic) provide access to state-of-the-art AI models, there is growing demand for more sovereign AI solutions. Organizations and nations increasingly seek to develop and deploy models on hardware that they control, rather than depending solely on a handful of proprietary companies for AI models and infrastructure [16, 37].

Organizations ranging from universities and national laboratories to supercomputing centers and small enterprises have made significant capital investments in GPU resources [11, 28, 69]. However, scaling centralized infrastructure to support the demands of next-generation AI workloads has significant logistical barriers, such as provisioning sufficient power to build even larger-scale facilities. Hence, initiatives like the EU AI Factories [11], the U.S. National AI Research Resource (NAIRR) [18] and Japan's AI Bridging Cloud Infrastructure (ABCI) [38] aim to connect multiple geo-distributed GPU-based High Performance Computing (HPC) clusters into a unified computing platform for training and serving large-scale AI models. The Swiss AI initiative [52] already uses the Alps supercomputer [33], which consists of multiple GPU-based HPC clusters, to train and serve open foundation models like Apertus [20]. Such initiatives are shaping the AI infrastructure landscape, evolving isolated hardware clusters into federated computing platforms.

Offloading model serving to commercial model-as-a-service cloud platforms (e.g., Together.AI, Fireworks.AI), which automate inference for open-weight and custom user models, is not sufficient, as they do not meet key requirements for sovereign AI initiatives and researchers: (**1**) *Cost-efficiency at large scale:* High-volume inference, such as for synthetic data generation or extensive evaluation, is often cost-prohibitive on public clouds. Organizations that have already invested heavily in local GPU clusters (often composed of legacy hardware) find it significantly more cost-effective to harness on-premise resources. (**2**) *Privacy and data sovereignty:* Strict data privacy regulations often preclude the use of external commercial services. For example, some medical research involving sensitive patient data or corporate workflows involving proprietary IP cannot be processed on third-party platforms without risking compliance violations.

Thus, there is a pressing need to serve large-scale models directly on sovereign, federated infrastructure, harnessing the heterogeneous hardware resources that organizations already own. These existing hardware clusters often have many hetero-

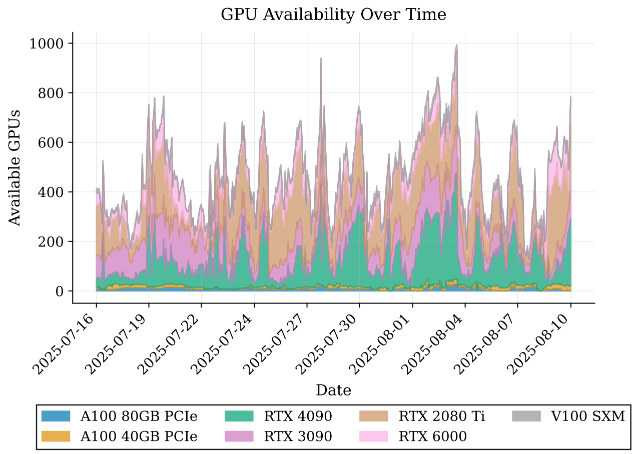


Figure 1: Number of available, idle GPUs in an HPC cluster from a research institution.

geneous and frequently under-utilized nodes. Figure 1 shows an example of a university cluster in which the availability of GPUs, ranging from legacy hardware to state-of-the-art accelerators, varies significantly over time. In this paper, we explore how to treat these heterogeneous, federated clusters as fungible resources for large-scale model serving. By pooling these distributed resources, we aim to absorb the volatility of individual cluster usage, effectively converting sporadic idle capacity into a usable serving platform.

A key challenge in using federated HPC clusters for model serving is that HPC environments are designed for long-running, static batch jobs, rather than the interactive, latency-sensitive workloads typical of model serving. Furthermore, modern LLM serving engines (e.g., vLLM [27]) do not have native orchestration capabilities, such as service discovery, load balancing, autoscaling — they rely on cloud-native orchestration platforms such as Kubernetes [44]. Unfortunately, deploying cloud-native stacks in HPC environtments is often logistically difficult (due to strict user-privilege limitations) or technically infeasible (due to the complexity of integrating with legacy HPC software stacks). Consequently, researchers are often confined to batch schedulers (e.g., Slurm [62]) that prioritize fairness over low latency and elasticity, and lack primitives for service discovery and load balancing.

To bridge this gap, we present LLMFabric, a serving system designed to harness existing HPC clusters across universities and research institutes to enable efficient serving of large ML models. LLMFabric is a thin layer on top of existing model serving engines (e.g., vLLM [27], SGLang [66]), as well as well-established HPC resource management utilities (e.g., Slurm [62]), providing a unified, user-friendly interface for researchers to serve and access a wide range of ML models. Under the hood, LLMFabric constructs a mesh of heterogeneous, distributed resources. Users can contribute LLM serving engine instances running on their local hardware and serving custom models. They can also access models hosted by other users on hardware across the mesh. This design enables cross-institutional sharing of hardware resources, as well as deployed serving engines, while providing a simple, unified API for users to interact with the system.

At the time of writing, LLMFabric has been deployed and operational for roughly 16 months. Thus far, it has served over 13 million requests and processed over 15 billion tokens across 142 different models, for over 500 researchers across multiple institutions. As a research project rather than a commercial service, LLMFabric does not currently provide strong availability guarantees.

We believe LLMFabric is interesting to the system research community for several reasons. (**1**) It is one of the first deployed and operational systems that enables cross-institutional sharing of hardware resources, which presents unique challenges in many system research areas, such as resource placement, scheduling and performance optimization. We present and open-source our design and implementation of the system. (**2**) We will make the traces collected from the system publicly available, with over 13 million requests and 15 billion tokens processed over a period of one year and 142 different models. We believe these traces can facilitate future research in this area. (**3**) We describe and discuss practical techniques and lessons learnt in order to operate such a system in an efficient manner. We hope LLMFabric can serve as a blueprint for other universities and research institutes to build their own ML model serving systems, fostering collaboration and resource sharing in the research community.

2 Background

We discuss the system environment in today's HPC clusters (which provide the backbone for sovereign AI infrastructure) and contrast their current design with the requirements of modern LLM serving engines.

2.1 HPC Environments

Most "AI Factories" and sovereign AI initiatives do not build entirely new infrastructure. Instead, they typically extend existing HPC clusters with AI hardware accelerators. These clusters are organized around a classic HPC software stack: a batch scheduler such as Slurm [62] manages job queues and allocates nodes to long-running jobs, prioritizing raw throughput and batch processing over interactivity [31].

Infrastructure operators aiming to run LLM inference in these environments face a dilemma between two imperfect operational models: (**1**) The most common strategy is to maintain the traditional HPC stack (e.g., Slurm [62]) for all workloads. While this unifies system administration, it prevents the effective deployment of modern inference workloads [31], which require a persistent service identity, elasticity, and other features incompatible with the static, queue-based nature of HPC job scheduling. (**2**) An alternative approach is to isolate a dedicated portion of the cluster to run a separate cloud-native stack (e.g., Kubernetes [44]) alongside the HPC partition. While this provides necessary environment for model serving, it rigidly isolates the two resource pools. The two resource pools are isolated and unable to share idle capacity, leading to resource fragmentation and underutilization. Although recent research proposed an integrated dual-stack architecture [31], such a solution remains complex and, to our knowledge, is rarely deployed in production environments. This leaves researchers confined to the legacy batch scheduler, creating a gap for a lightweight solution that can bridge modern serving engines with HPC constraints.

2.2 LLM Serving Engines and the Gap

LLM inference involves two distinct phases: *Prefill* and *decoding*. The prefill phase processes input prompts to populate the initial KV cache, while the decoding phase generates tokens auto-regressively, updating the KV cache per step. This duality creates unique resource demands: Prefill is typically compute-bound and highly parallelizable across the sequence length, whereas decoding is memory-bound due to the low arithmetic intensity of auto-regressive generation. In a multi-tenant research environment, this complexity is compounded by multi-model serving, where users request a diverse catalog of models ranging from standard open-weights LLMs (e.g., Llama-3) to specialized, fine-tuned variants. Unlike commercial services that serve a few hot models, research workloads are characterized by a long tail of bespoke models (see Figure 7), requiring the system to be able to efficiently manage and route requests across this heterogeneous model landscape.

To address these computational constraints at the instance level, high-performance inference engines such as vLLM [27] and SGLang [66] have emerged. These engines employ advanced techniques like PagedAttention and continuous batching [63] to maximize the throughput of single-instance execution. However, they are designed primarily as standalone runtime engines: In commercial cloud environments, these engines serve merely as the data plane, relying on an external orchestrator (typically Kubernetes [44]) to handle the control plane responsibilities: service discovery, load balancing, fault tolerance, auto-scaling, etc.

However, in the HPC environments described in §2.1, this necessary orchestration layer is missing. A researcher running `vllm serve` inside a Slurm job receives a transient endpoint bound to a specific compute node, with no mechanism to route external traffic to it or recover from preemption. While the engine optimizes the *inference* (matrix multiplication and memory management), it does not provide a *service*. To build a scalable, multi-tenant serving system on HPC infrastructure, a new abstraction layer is required: One that provides "cloud-native" orchestration features entirely within user-space, without requiring root privileges or cluster partitioning.

2.3 System Requirements

To bridge the gap between modern LLM serving engines and legacy HPC infrastructure, a serving system for the research community must meet key requirements:

R1: Abstraction over heterogeneous engines. Researchers often experiment with a wide range of models, including open-source models, and sometimes their own customized, fine-tuned versions. Unlike commercial systems that typically focus on a few popular models, a research-oriented serving system must accommodate this diversity and provide support for various model architectures and configurations. Meanwhile, different research groups, based on their needs and expertise, prefer different serving systems. Different serving systems may offer unique features (e.g., the ability to handle specific model types, performance optimizations, or hardware compatibility). For example, Turbomind [65] supports V100 GPUs while SGLang [66] deprecates support for those GPUs. A one-engine-fits-all approach may not be effective in such an environment. *The system must decouple the user-facing API from the execution backend, functioning as a meta-layer that can transparently route requests to the most appropriate engine (e.g., vLLM, TGI, TurboMind) based on the underlying hardware capabilities.*

R2: Non-invasive, user-space orchestration. Scaling inference beyond a single instance requires an orchestration layer to handle service discovery, load balancing and elasticity. In commercial clouds, this is the domain of Kubernetes [44]. However, deploying Kubernetes on shared HPC clusters is often administratively impossible (requiring root privileges) or technically infeasible (conflicting with existing batch schedulers). Researchers are proficient with Slurm [62] but often lack the specialized DevOps expertise to manage complex distributed cloud-native infrastructure. *The system must provide "cloud-like" orchestration features entirely in user-space. It must operate as an overlay on top of standard batch schedulers, allowing researchers to spin up serving clusters using standard permissions without requiring administrative intervention or kernel-level modifications.*

R3: Fault-tolerance for federated, fungible resources. Unlike dedicated commercial clouds that offer service level agreement-backed availability, federated academic resources often have volatile availability and may be down for hours due to maintenance. Access is frequently granted via "scavenger" or "spot" queues, where resources can be preempted for higher-priority tasks, while the underlying network infrastructure introduces latency variance and potential partitions. *The system must implement a fault-tolerant control plane that abstracts away physical location and stability constraints. It must dynamically route traffic and handle sudden node preemption or network failures without service interruption.*

3 LLMFabric System Design

To meet the requirements in §2.3, we categorize the users into two roles: *Service providers* and *consumers*. Service providers are users who deploy their models to the system, while consumers are users who use the models served by the system through APIs. A single user can act as both a service provider and a consumer simultaneously. This design contrasts with other commercial serving systems where models are typically deployed by a centralized team and consumed by end

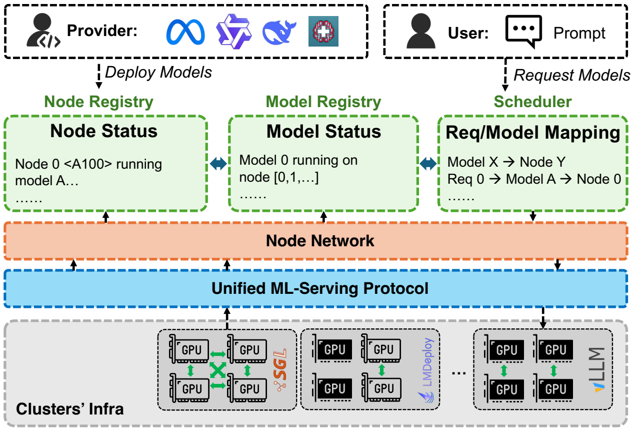


Figure 2: System architecture of LLMFabric Serving.

users. In this section, we present the design of LLMFabric, a multi-tenant LLM serving system optimized for decentralized research environments. We first outline the system architecture, component details, and an example workflow illustrating how the system operates in practice.

3.1 System Overview

System architecture. Figure 2 illustrates the multi-layered architecture of LLMFabric. At the bottom layer lies the *heterogeneous GPU infrastructure*, which aggregates compute resources from multiple clusters. These GPUs may have diverse hardware characteristics (such as memory capacity, processing power, and network bandwidth) and software environments (such as serving frameworks, cluster managers or orchestration tools, CUDA versions, etc).

Service protocol. To manage this heterogeneity, LLMFabric relies on a *unified serving protocol*. We adopt and extend the OpenAI-compatible API specification as our common interface. This abstraction layer decouples the higher-level system operations from the underlying execution environments. The protocol standardizes the interface between the control plane and the serving engines, ensuring that all nodes, regardless of their hardware or software stack, expose a consistent API and request format. Consequently, users interact with a consistent, familiar set of endpoints, while the system transparently routes requests to the appropriate hardware.

Node network. To route requests to nodes[1], LLMFabric has a mechanism for service discovery that allows nodes to discover peers and exchange metadata without a central coordinator. LLMFabric relies on a decentralized peer-to-peer network. To maintain a consistent global view, LLMFabric represents the network as a Growth-only Map (G-Map) Conflict-free Replicated Data Type (CRDT) [47]. The map key is a randomly generated session ID assigned to the node upon startup, and the value is a tuple containing the node's lifecycle state and its associated metadata payload.

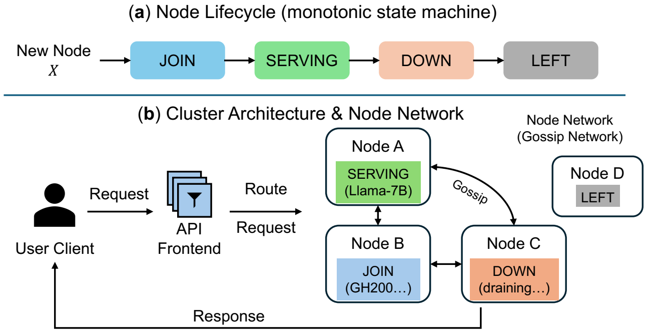


Figure 3: Workflow example of LLMFabric.

A node transitions through a strictly ordered lifecycle: JOIN $\prec$ SERVING $\prec$ DOWN $\prec$ LEFT. This monotonic state progression drives the evolution of the node's metadata: 1) JOIN: The node registers with the network. At this stage, the metadata may be generic (e.g., hardware specs only), allowing providers to join without pre-committing to specific models. 2) SERVING: The node commits to a serving configuration and updates its metadata to include the specific list of models being served. Since our merge function $\sqcup$ selects the entry with the highest lifecycle state (e.g., SERVING overwrites JOIN), the network eventually converges to the most recent, fully specified configuration for every node, thus maintaining a global view of available resources.

Provider/Model registry. By aggregating the metadata propagated through the node network, the nodes collectively maintain a distributed *Provider/Model Registry*. This component exposes the real-time *model/node status* as a queryable index of the physical infrastructure. Instead of relying on a centralized catalog, the registry allows the scheduler or users to query the global state to discover nodes based on specific criteria (e.g., filtering for nodes with A100 GPUs or specific VRAM thresholds) directly from the decentralized mesh.

Service scheduler. While users can manually assign nodes to serve specific models, LLMFabric also has a centralized *service scheduler* that can automatically optimize model placement to balance performance requirements with resource availability and cost efficiency. We describe its scheduling policy in §3.3.

API frontend. LLMFabric uses an API frontend as the gateway between external clients and the node network. This component maintains synchronous connections with users, managing authentication, usage tracking and the complete request-response lifecycle.

User interface. To ease access, LLMFabric provides a web-based dashboard that allows users to browse the real-time catalog of actively served models alongside their hosting hardware, manage API keys, and validate model behavior through an interactive chat interface. This design lowers the barrier to entry and encourages adoption among researchers.

3.2 Workflow Example

Figure 3 illustrates an example workflow of LLMFabric for two types of requests: *node registration* and *model inference*.

[1] A *node* is a logical compute unit (e.g., 1 or more GPUs) that implements the OpenAI-compatible interface and independently processes requests.

Life of a node. When a provider contributes a node to LLMFabric, it initiates the `JOIN` process by contacting a known *bootstrap peer*, defined in the cluster configuration. LLMFabric employs a randomized gossip protocol for membership management: Upon connection, the node exchanges state with the bootstrap peer, and this membership information propagates rapidly across the network. Within a few gossip rounds, the system converges, and all peers update their local registries to include the new node.

Life of a model/service. Once a node has joined the network, it begins the lifecycle of a model/service. The node deploys a specific model (either explicitly configured by the provider or dynamically assigned by the service scheduler). To announce availability, the node updates its local CRDT state with a service record containing the model identifier, the node's unique *session ID*, and optionally the *provider ID*. This metadata is automatically replicated to all peers, enabling other nodes to discover the new service endpoint and include it in their routing tables.

Life of an inference request. When a consumer sends an inference request through the API frontend, it is routed to an ingress node. To schedule the request, the ingress node queries its local CRDT replica to identify candidate nodes that are serving the requested model. LLMFabric routes requests based on the unique *session ID* and *provider ID* associated with each node. The system filters the initial candidate set against the user's specified *trusted providers*. Any node whose provider ID is not in the user's allowlist is pruned from the candidates pool. From the remaining authorized candidates, a specific session ID is selected via a load-balancing strategy, and the request is dispatched to that node's API endpoint. This decentralized approach ensures high availability and fault tolerance, as the system can dynamically route traffic across all compliant nodes, continuing to function even if individual nodes go offline.

Fault tolerance. In case a model process or entire node fails, LLMFabric detects the failure and continues operating the inference service without the failed process or node. A per-node LLMFabric process continuously monitors the health of each model serving process. If the process terminates unexpectedly, the node explicitly updates its state to the `DOWN` state in the network, signaling to peers that the model is unavailable. To handle node-level failures (e.g., if the LLMFabric process itself crashes or the host fails), the network utilizes peer-to-peer heartbeats. If a node ceases to send heartbeats, peers detect the silence within seconds and update their local registry to mark the node as `LEFT`. This rapid propagation ensures the scheduler ceases routing traffic to failed nodes, effectively isolating faults while the system attempts to re-allocate resources.

3.3 Model Placement and Scheduling

LLMFabric allows users to manually specify which nodes in the cluster will serve which models. However, these decisions can be difficult for users to make. Users will often request the highest-end GPUs for their jobs to maximize performance, but these devices are usually also the most expensive and their availability is often scarce due to high demand. Therefore, LLMFabric also allows users to defer the decision of which models should be hosted on which nodes to the system. We formulate model deployment as an optimization problem minimizing overall system latency.

Consider a heterogeneous cluster $\mathbf{D} = \{d_1, \ldots, d_N\}$, where d_n denotes the count of available GPUs of type n. We serve M distinct model types, indexed by m, with incoming workloads $\mathbf{W} = \{w_1, \ldots, w_M\}$. Each workload w_m is defined by a tuple $\langle \lambda, \mu_{in}, \mu_{out} \rangle$ representing arrival rate and input/output length distributions, which are collected and monitored by the API frontend. The scheduler determines two key decision variables:

1. **Allocation matrix ($\mathbf{A} \in \mathbb{R}^{M \times N}$):** Where $a_{m,n}$ is the number of type-n GPUs allocated to model m.

2. **Parallelism strategy ($\mathbf{P} = \{p_1, \ldots, p_M\}$):** Where p_m defines the data and tensor parallelism configuration for model m.

The optimization objective is formulated as:

$$\min_{\mathbf{A},\mathbf{P}} \quad L(\mathbf{W}, \mathbf{A}, \mathbf{P}) \tag{1}$$

where $L(\cdot)$ represents the estimated system latency.

Solving Eq. 1 requires an efficient method to evaluate $L(\cdot)$ for any candidate configuration. The typical approach of profiling jobs on different hardware configurations is accurate but not scalable, particularly for clusters that may have highly heterogeneous hardware, such as the cases we consider. Instead, we use a *serving simulator* to analytically estimate performance.

The serving simulator takes model specifications (e.g., architecture, parameter count), hardware characteristics (e.g., GPU type) and request workloads (e.g., arrival times, input/output lengths) as inputs. The simulation operates in two stages, extending the analytical method of `LLM-Viewer` [64] from single-request analysis to multi-request serving scenarios. (**1**) We first enumerate all operators in the model, such as the $\mathbf{q}, \mathbf{k}, \mathbf{v}$ linear projections, self-attention computations, feed-forward layers, etc. For each operator, we estimate its execution time on the target hardware. In our current design, we estimate execution time using the roofline model [4, 59], taking into account the operator's computational intensity and the hardware's peak performance and memory bandwidth. This analytical model, though not as accurate as empirical profiling, provides a unique advantage in our environment as it does not require offline profiling, making it adaptable to new models, new hardware, and diverse serving environments, such as model or KV cache quantization, varying sequence lengths and batch sizes, etc. (**2**) In the second stage, we simulate the concurrent execution of the input workload. Based on the operator latencies derived in the first stage, the simulator models complex system behaviors including continuous batching, model parallelism, and scheduling policies. The simulator outputs key metrics like latency and throughput, which drive the service scheduler's deployment decisions.

These simulation results parameterize our objective function, allowing us to evaluate the specific trade-offs inherent in Eq. 1. The scheduler uses *Constraint Programming* (CP) [45] to solve the optimization problem, which excels at exploring discrete combinatorial spaces through constraint propagation, making it ideal for our resource allocation problem. We enforce three key constraints to ensure feasible deployments: *C1: Global Resource Capacity.* The total allocation for each GPU type n cannot exceed availability:

$$\sum_{m=1}^{M} a_{m,n} \leq d_n, \quad \forall n. \tag{2}$$

C2: Memory Feasibility. To prune the search space, we ensure any assigned GPU type n has sufficient memory (mem_n) to host the model weights and KV cache (M_m):

$$a_{m,n} \cdot \text{mem}_n \geq M_m, \quad \forall (m,n) \text{ where } a_{m,n} > 0. \tag{3}$$

C3: Parallelism compatibility. For each model type m, the parallelism strategy p_m specifies a configuration $\langle \psi_{\text{DP}}^{(n)}, \psi_{\text{TP}}^{(n)} \rangle$ for each GPU type n where $a_{m,n} > 0$, representing the data parallelism [30] and tensor parallelism [36, 49] degrees, respectively. The configuration must satisfy:

$$a_{m,n} = \psi_{\text{DP}}^{(n)} \times \psi_{\text{TP}}^{(n)}, \quad \forall (m,n) \text{ where } a_{m,n} > 0, \tag{4}$$

ensuring that the $a_{m,n}$ GPUs of type n allocated to model m are exactly partitioned into $\psi_{\text{DP}}^{(n)}$ data-parallel replicas, each employing $\psi_{\text{TP}}^{(n)}$-way tensor parallelism.

4 Implementation

Our implementation of LLMFabric consists of approximately 7K lines of Go. We choose Go because of its strong support for concurrency, cross-platform compatibility, and a rich ecosystem of libraries for building distributed systems. LLMFabric uses `libp2p` [29], a modular network stack, to construct the peer-to-peer network that underpins the decentralized architecture. LLMFabric provides a `Gin` [14]-based API endpoint that exposes RESTful endpoints for accessing the internal status and functionalities of the system. The complementary API frontend is implemented using `FastAPI` [13].

LLMFabric is distributed as a pre-built command-line tool. To deploy a model, users simply encapsulate their existing inference commands (e.g., `vllm` or `sglang`) within the LLMFabric CLI. For instance, a user can start a node by running: `llm-fabric start --process vllm serve [model_name]`. Alternatively, users can start a node without a pre-defined model by running `llm-fabric start`. In this mode, the node registers its available hardware resources with the network, allowing the service scheduler to assign models to the node later.

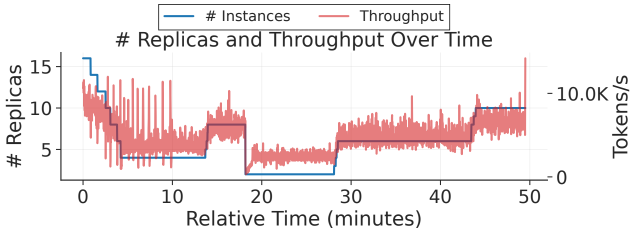


Figure 4: The system's ability to recover from node failures over time.

Upon initialization, LLMFabric first runs the specified command as a child process if provided. Once the subprocess starts successfully, LLMFabric notifies the node network about its presence, as well as the metadata about hardware specifications (e.g., GPU type, VRAM capacity). In the meantime, the current LLMFabric node starts to synchronize with other nodes in the network through the gossip protocol to learn about other existing nodes and their states. LLMFabric then monitors the subprocess by periodically checking its health status, and once the subprocess's `/models` endpoint becomes available, LLMFabric reads the model metadata from the endpoint and broadcasts the information to all nodes in the network.

Under the hood, LLMFabric assigns a unique identifier to each deployed node, and this identifier will be re-generated if the node restarts. Each node then can be specified by *multi-address*, which contains the transport protocol (e.g., TCP/UDP), port number where LLMFabric is listening, as well as the unique node identifier. To discover all available nodes in the network, LLMFabric leverages *Kademlia DHT* (KAD) [34], a distributed hash table protocol for peer addressing and discovery.

5 System Evaluation

In this section, we evaluate the fault-tolerance and performance of LLMFabric with benchmark workloads, while in §6, we will analyze the real workloads on the deployed system. Our evaluation aims to answer the following key questions:
Q1: Elasticity and fault-tolerance. Can LLMFabric maintain service continuity and scale throughput proportionally in a dynamic environment with node failures and additions?
Q2: Placement efficiency. How does our heterogeneity-aware placement strategy compare to standard heuristics in terms of aggregate cluster throughput?
Q3: Simulation accuracy. Does the serving simulator accurately capture relative hardware performance differences to guide scheduling decisions?
Experiment setup. We conduct our experiments on a heterogeneous environment composed of NVIDIA GPUs, ranging from datacenter-grade accelerators (GH200, H100, A100) to consumer-grade hardware (RTX 3090). This diversity al-

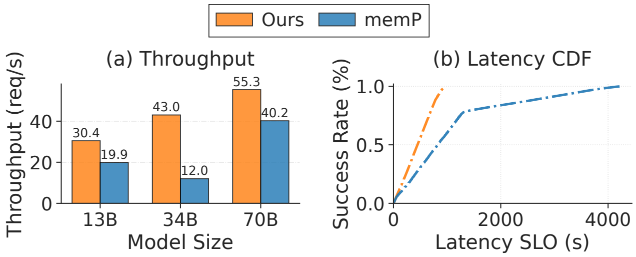


Figure 5: Throughput and Latency comparison between our placement method and memory-proportional approach.

lows us to evaluate LLMFabric's ability to handle hardware heterogeneity. We use LLMFabric to serve popular open-source LLMs with varying parameter sizes: `Llama-2-13b`, `CodeLlama-34b`, and `Llama-3.3-70B-Instruct`. Unless otherwise stated, request arrival times follow a Poisson process, with input and output token lengths sampled from Normal distributions derived from real-world traces collected from our deployment (see more details about traces in §6).

Elasticity and fault-tolerance. To evaluate LLMFabric in a dynamic HPC-based environment, where nodes may join or leave the network at any time, we deploy LLMFabric on a system with at most 64 GH200 GPUs, serving a continuous stream of inference requests while varying the available worker pool by randomly terminating nodes in the system during the experiment. Figure 4 reports the throughput (in tokens/s) alongside the number of active replicas over time. LLMFabric's throughput (red line) adapts proportionally, on average, to the number of active replicas (blue line), showing that LLMFabric can effectively adjust to changes in compute capacity without manual intervention. Furthermore, despite the abrupt termination of nodes, we observed zero user-facing HTTP errors during the entire experiment. This confirms that our decentralized registry maintains service continuity even under high churn rates, successfully routing requests to remaining healthy nodes while the cluster topology stabilizes.

Placement strategy. We evaluate the effectiveness of our placement strategy by comparing it against a resource-aware heuristic policy. As no existing placement method considers both hardware and model heterogeneity, we implement a memory-proportional policy, *memP*, inspired by HexGen [25] that represents a standard heuristic often applied in our environment. It allocates GPU resources proportional to the memory demand of different models, defined as the product of the average request arrival rate and the model parameter size. We synthesized a workload modeled from real-world traces collected from LLMFabric, consisting of three distinct models with varying sizes and load characteristics to simulate a realistic, heterogeneous environment: `Llama-2-13b`, `CodeLlama-34b`, and `Llama-3.3-70B-Instruct`. Requests for each model follow a Poisson arrival process with average rates of 110, 185.5, and 221 req/s, respectively. We sample input and output sequence lengths from Normal distributions, with mean input lengths ranging from approximately 600 to 1170 tokens, and mean output lengths ranging from 64 to 530 tokens. Table 1 and Figure 5 show the impact of placement decisions on cluster performance. The baseline approach prioritizes the largest model (70B) by over-provisioning it with GH200 and A100 GPUs. However, this creates a resource imbalance that starves the 34B model of high-performance compute, resulting in suboptimal aggregate performance. In contrast, our heterogeneity-aware strategy balances allocation across model and GPU types. We do not claim that our policy is globally optimal. Our goal here is simply to highlight the critical importance of placement decisions: Explicitly modeling system heterogeneity yields better resource utilization than simple heuristics that are often used in practice.

Table 1: Placement configurations for different models of LLMFabric and the memory-proportional approach.

Method	13B	34B	70B
MemP	4×A100	16×A100	32×GH200+4×A100
Ours	6×A100	16×GH200+10×A100	16×GH200+8×A100

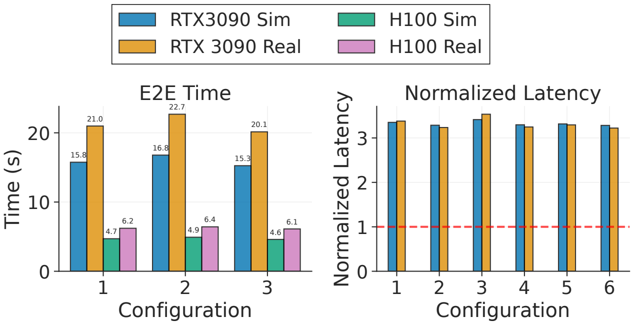


Figure 6: End-to-end performance comparison between H100 and RTX 3090 GPUs. (Left) Absolute latency in seconds. (Right) Latency normalized to the H100 (lower is better); the RTX 3090 exhibits 3.2×–3.4× higher latency than the H100 across configurations.

Simulator validation. Figure 6 compares the end-to-end latency estimations for workloads running on H100 and RTX 3090 GPUs, with different input and output lengths. Although we find the absolute latency estimations deviate from real measurements by 23% to 26%, mainly due to the accumulated errors from decode time estimations, the simulator accurately captures the relative performance differences between the two GPUs across different input/output lengths. This shows that the simulator can effectively differentiate hardware performance and can serve as a useful tool for making informed, heterogeneity-aware scheduling decisions in LLMFabric.

6 Trace Analysis

We have been operating LLMFabric for over 16 months as an LLM serving platform for AI researchers across multiple academic institutions participating in a large AI initiative.[2] All users accessing LLMFabric must register and obtain an API key using their institutional login credentials. At the API frontend, we log all incoming requests. For each request, we

[2] We omit the name of the initiative for double-blind review.

Trace	Span	Schema	# Models	LLM
BurstGPT [57]	121 days	T, M, I/O	2	✓
ShareGPT	-	M, I/O	multiple	✓
Azure 1 [39]	days	T, I/O	1	✓
Azure 2 [50]	weeks	T, I/O	1	✓
Azure 3 [42]	weeks	T, I/O	1	✓
Mooncake [41]	-	-	1	✓
ChatLMSys [17]	days	M, I/O	4⋆	✓
Azure MAF [46]	days	T, M⋆	multiple⋆	✗
Ours	16 months	T, M, I/O	46/142	✓

Table 2: Trace characteristics summary. T: timestamp, M: model, I/O: input/output tokens, ⋆: not inference, just function invocations. Number of models only counts unique models.

record a timestamp, the model identifier, aggregate input and output token counts, and sampling parameters (e.g., temperature, maximum tokens).

6.1 Trace Overview

We plan to release the anonymized trace dataset collected from July 2024 to October 2025 to facilitate future research on AI serving system infrastructure. This dataset includes over 13 million requests, with over 15 billion tokens processed. Due to privacy and compliance requirements, we neither use nor disclose the actual content of both input prompts and output responses in our traces. However, we do analyze prefix sharing and token reuse characteristics using the approach first described in Mooncake [41], which involves irreversibly hashing the input tokens and putting 16 consecutive tokens into a single hash bucket. A bucket ID will be equal to another if and only if the 16 tokens in both buckets are identical. We hash the bucket ID so that no information can be inferred from the ID. Hence, our trace does not include any user-identifiable information. We still allow users to bypass the usage tracking in cases where it is necessary to ensure data compliance.

In Table 2, we summarize key differences between our traces and existing public traces. Most existing traces focus on a single or very few models (often a popular closed-source LLM such as GPT-3.5 or GPT-4) while our trace includes a diverse set of real-world models, with over 142 different models being requested. Among those models, 46 are existing open-weight models that are publicly available, while the remaining 96 models are custom-trained models developed within our researchers and users community. Additionally, our traces span a longer period of time, allowing us to analyze long-term trends in model usage and request patterns.

6.2 Workload Characterization

Overall model usage. The workload in our trace exhibits a multi-tier power-law distribution. Figure 7 shows the total token consumption of the top 20 models, including both input and output tokens per model, ranked from highest to lowest. The first tier consists of the top 3 models, which account for a significant portion of total token usage. These models are general-purpose LLMs that are used for various tasks where a specific model is not required. the second tier includes several models that includes both multi-modal models (notably *Qwen-2.5-VL*) and also some particular models that users are interested in exploring and evaluating (e.g., *Apertus* models, which are trained within the SwissAI community [20]). The third tier consists of a long tail of models that are used infrequently, often for specialized tasks or by niche user groups.

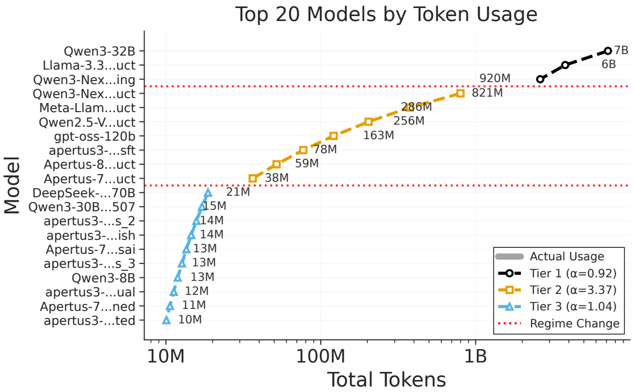


Figure 7: Overall model token usage. Each bar represents the total token consumption (input + output) for a specific model, ranked from highest to lowest usage.

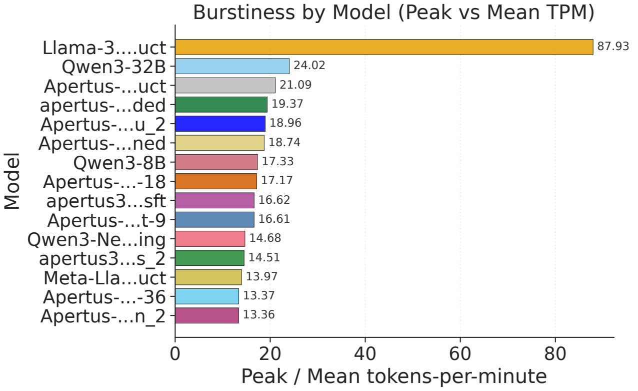


Figure 8: Peak-to-average request rate ratio per model during their active minutes.

> *Key Takeaway 1*: Serving systems must support a skewed workload distribution, effectively handling both popular general-purpose models and a long tail of diverse, niche models required by specific user communities.

Per-model request burstiness. Next, we characterize model request burstiness along two dimensions: Magnitude and temporal dynamics. We define the *instantaneous burstiness* as the ratio of the instantaneous tokens per minute (TPM) to the model's average TPM. We then quantify the overall *burstiness magnitude* as the peak of this ratio. We compute the average TPM strictly over active minutes (minutes with at least one request) to avoid artificially inflating the ratio due to periods of inactivity. Figure 8 reports the magnitude, ranking models by their burstiness magnitude to highlight the most ex-

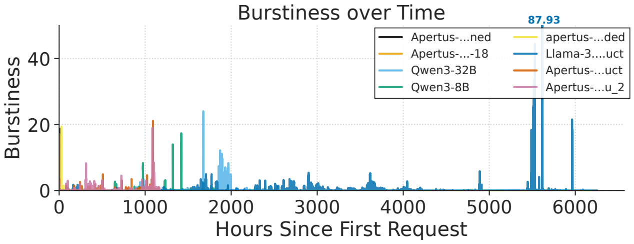


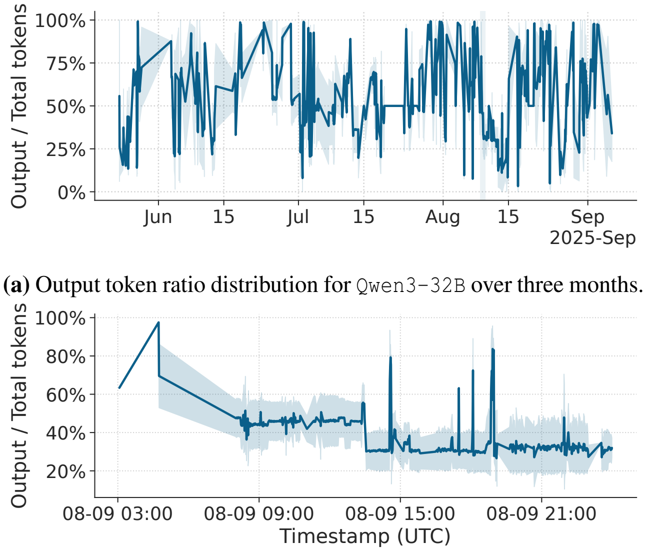
Figure 9: Burstiness over time for several representative models. Burstiness is defined as the ratio of instantaneous TPM to the model's average TPM over active periods. Timelines are aligned to the first request at $t = 0$.

treme bursty workloads. Overall, we observe extreme volatility across the board. Even among the top 15 most used models, the lowest burstiness exceeds 13, while the most bursty model reaches a ratio of over 80. This indicates that average request arrival rates significantly underestimate peak loads. For example, Figure 9 shows the temporal request invocation dynamics for a variety of models, with timelines aligned to each model's first request ($t = 0$). This reveals when load spikes occur relative to the start of a model's operation.

Key Takeaway 2: Dynamic resource allocation and elastic scaling are essential, as static provisioning based on average rates does not address the extreme burstiness and synchronized demand surges of real-world model usage. In the meantime, provisioning for peak usage would lead to large periods of underutilization.

Input / Output ratio characteristics. A request's resource footprint depends heavily on its structure: Input-heavy requests strain compute during the prefill phase, while output-heavy requests consume memory bandwidth during the decode phase. Therefore, we analyze how the output sequence lengths of requests compares to the total (input + output) sequence lengths. Figure 10 plots the output token ratio for `Qwen3-32B`, over three months and zooming into the busiest day during this time period. As shown in Figure 10a, the workload is far from uniform: The ratio of output tokens to total tokens fluctuates wildly over a three-month period. A fine-grained view of the busiest day (Figure 10b) reveals a bimodal behavior: A stable baseline where output tokens constitute 30% to 40% of the total tokens, frequently punctuated by generation-heavy spikes exceeding 80%. We observed similar patterns in other models as well. This variance makes classic continuous batching strategies insufficient, as compute-bound prefills frequently collide with memory-bound decodes, causing significant interference.

Key Takeaway 3: Interference between compute-bound prefill and memory-bound decode stages is a critical issue in production LLM serving. This highlights the importance of optimizations such as chunked prefill [2] and prefill-decode disaggregation [23, 39, 51, 67] to mitigate this interference.



(a) Output token ratio distribution for `Qwen3-32B` over three months.

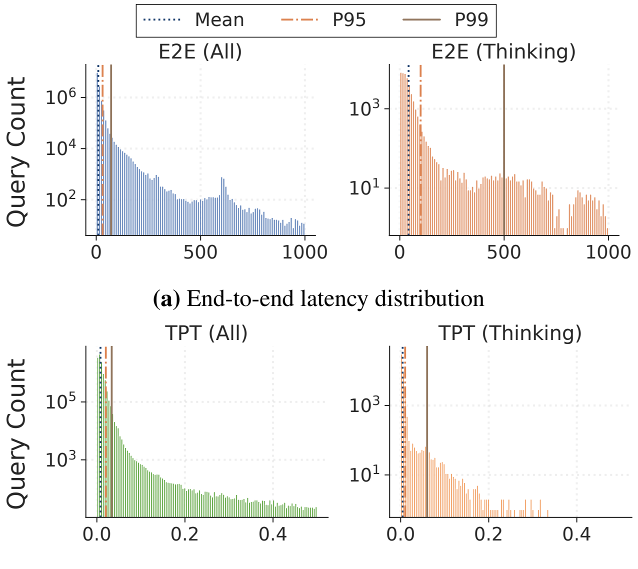
(b) Output token ratio distribution for `Qwen3-32B` in the busiest day.

Figure 10: Output token ratio distribution for `Qwen3-32B`.

Latency distribution. Figure 11 highlights a decoupling between generation speed and total latency. While the Time Per Token (TPT) remains comparable between models (Figure 11b), the End-to-End (E2E) latency diverges sharply (Figure 11a). The reasoning model exhibits a heavy tail, with a P95 latency of 99.80s, more than 3× higher than the overall average of 30.44s. This discrepancy confirms that the bottleneck is not the decoding speed, but the sheer volume of hidden reasoning tokens generated before the final answer.

Key Takeaway 4: Service Level Objectives (SLOs) must be differentiated to accommodate reasoning models, applying distinct timeout and retry policies that account for their significantly higher latency tails compared to standard models. To actively guarantee SLOs, systems could implement "reasoning budget" [32, 58] that dynamically caps the computational depth of reasoning traces to fit within time windows, or SLO-aware model selection [6, 24] where the system dispatches queries to the appropriate models, or SLO-aware model serving [7].

Prefix sharing characteristics. Beyond the structure of individual requests, we investigate the computational redundancy *across* requests. We analyze the token reuse ratio, defined as the fraction of input tokens overlapping with recent history, to identify opportunities for avoiding re-computation. Figure 12 reveals a distinction in usage patterns between model types. The reasoning-heavy *Qwen3-Next-80B-A3B-Thinking* exhibits a massive reuse ratio exceeding 90%. This implies that the dominant workload for this reasoning model consists of deep, iterative multi-turn sessions where users build progressively on a shared context. In contrast, the standard *Instruct* variant shows a modest reuse ratio of $\sim$ 15%, reflecting the usage pattern of more independent, single-turn queries. This suggests that much of the prefill computation in reasoning models may be redundant. This motivates KV-



(a) End-to-end latency distribution

(b) Time per token distribution

Figure 11: Overall E2E Latency and TPT distributions.

cache management optimizations like `LMCache` [8], which can amortize the quadratic attention costs inherent in these long-context, iterative workflows.

Key Takeaway 5: Our analysis of *Qwen3-Next-80B-A3B-Thinking* suggests that reasoning models, particularly when used in iterative, multi-turn sessions, might exhibit high prefix reuse ratios. This suggests that optimizations focused on prefix caching and KV-cache management [8] could be particularly beneficial for amortizing the quadratic attention costs in these workloads.

7 Discussion

In this section, we reflect on our experiences designing, implementing, and operating LLMFabric in real-world environments. We then share some observations on emerging workload patterns driven by agentic applications and reasoning models. Finally, we discuss the limitations of our current design and outline directions for future work.

7.1 Lessons learned

Flexibility and interoperability as first-class citizens. We observed that researchers often have varying preferences for serving engines and hardware capabilities, and that the flexibility to adapt to different hardware and software environments is crucial for the deployment and adoption of LLMFabric. In the context of LLM serving, the widespread adoption of the unified OpenAI-compatible interface across these serving engines provided a standardized abstraction layer. This allowed us to design LLMFabric to be engine-agnostic by default. Furthermore, we evaluated whether Kubernetes-based orchestration would be a better fit for our use case. However, given our reliance on Slurm-based HPC clusters for GPU resources, we found that integrating with existing HPC infrastructure provided a more seamless experience for our users.

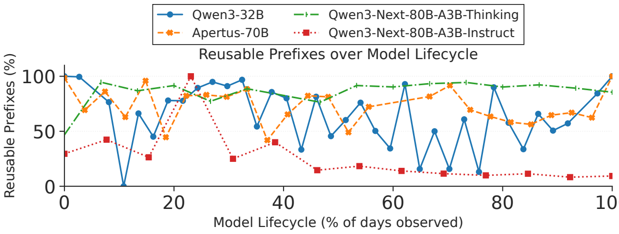


Figure 12: Reusable prefix ratio over model lifecycles for four representative models.

Need for more expressive abstractions. A unified interface introduces a hidden cost. While the OpenAI-compatible interface simplifies interaction, it obscures critical execution details like model quantization levels and context limits. This creates ambiguity when users have strict expectations regarding numerical precision and other parameters critical for model quality. To mitigate this, we currently require model providers to adhere to strict naming conventions (e.g., explicitly tagging quantization bits in model names) to ensure transparency. This experience underscores a deficiency in current LLM serving protocols: The lack of a protocol for negotiating model quality-of-service beyond simple identifiers.

Importance of proper systematic modeling. Early iterations of LLMFabric relied on a centralized database for tracking GPU worker availability. We found that enforcing strong consistency for metadata operations created a significant scalability bottleneck. The pivotal lesson was transitioning from ad-hoc state tracking to a formal algebraic model that represents the cluster state as a conflict-free replicated data type (CRDT). By modeling the system state as a monotonic lattice, we were able to decouple node interactions and adopt an eventually consistent model. This shift eliminated the central coordinator entirely, enabling the system to remain available even during network partitions.

Introspection, monitoring and human intervention. Debugging in a decentralized environment is inherently challenging due to the lack of direct access (e.g., SSH) to worker nodes. Consequently, we shifted our design philosophy from attempting to recover to fail-stop semantics. Instead of attempting complex runtime recovery, LLMFabric employs strict health checks, and upon detecting fatal anomalies (e.g., NCCL timeout), a node automatically evicts itself from the cluster to preserve global system integrity. To aid post-mortem analysis, we proactively collect logs to a sink before termination, ensuring that diagnostic context is preserved even when the node becomes inaccessible.

7.2 Workload-aware Serving Systems

We also conduct some preliminary experiments to explore how serving systems can become more workload and hardware-aware, to enable performance and energy optimization. In particular, since we notice that LLMFabric often serves reasoning and non-reasoning versions of models, we explore the serving system implications of reasoning while

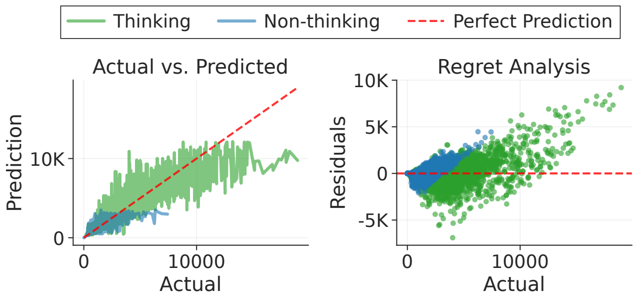


Figure 13: Agentic workloads characterization.

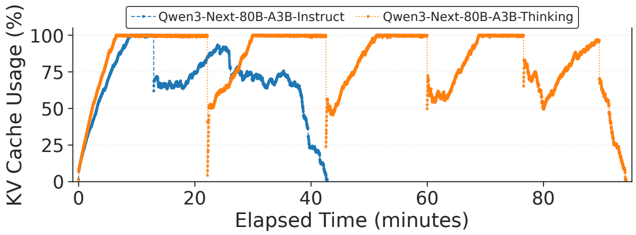


Figure 14: KV cache usage comparison between reasoning and non-reasoning models.

running representative public benchmarks.

Predictability of output sequence lengths. Accurately estimating output sequence length could enable the serving system to improve request scheduling and resource allocation decisions, as the ratio between output and input sequence lengths determines whether the workload is compute-bound (prefill dominates) or memory-bound (token generation dominates). We assess this predictability by replaying the CHESS [53] framework with BIRD test set [43] on `Qwen3-Next-80B-A3B`, comparing its standard (*Instruct*) and reasoning (*Thinking*) variants. Using a random forest regressor trained on agent metadata and input length, we observe a sharp divergence in predictability (Figure 13). While the non-reasoning model allows for more accurate prediction (MAE $\sim$ 94 tokens), the reasoning model exhibits high variability (MAE $\sim$ 700 tokens). *This suggests that the complexity of reasoning makes predicting execution time difficult.*

KV cache pressure. Reasoning models tend to produce longer outputs, which consumes additional memory. Figure 14 contrasts the KV cache usage under the same benchmark (*AIME 1983-2024* [56]. The non-reasoning model shows a dynamic usage with rapid reclamation cycles, finishing in $\sim$ 40 minutes. In contrast, the reasoning model extends execution to $\sim$ 95 minutes and exhibits a distinct sawtooth pattern: Sharp, sustained increases represent the accumulation of transient state during chain-of-thought reasoning, followed by sharp drops upon sequence completion. *The high cost of regenerating transient "reasoning" states requires sophisticated memory management, such as offloading KV caches to secondary storage during preemption rather than discarding and recomputing them.*

Energy consumption. The extended execution duration and high memory occupation of reasoning models observed above inevitably increase energy costs. Figure 15 quantifies this impact on A100 and GH200 architectures using the *AIME 1983-2024* benchmark [56]. We find that reasoning models are not just slower but also more energy expensive. The reasoning variant consumes over 2× the total energy of the non-reasoning baseline (rising from $\sim$ 2.0 kWh to $\sim$ 4.3 kWh). Crucially, the *energy cost per token* also jumps by $\sim$ 56% (from $\sim$ 0.16 to $\sim$ 0.25 kWh per million tokens). This non-linear scaling stems from a compound effect: The model generates more tokens (volume), and as the context window expands, the quadratic complexity of self-attention makes every subsequent decoding step more computationally intensive. Furthermore, we compare the energy consumption across two GPU architectures. While GH200s deliver superior raw performance, their energy consumption is on par with A100s for the same workload. We attribute this to two factors associated with this particular workload. First, the workload is memory-bound by the KV cache, restricting the batch sizes and preventing full saturation of GH200's compute units. Second, the sparse activation of MoE models exacerbates this by requiring more memory capactiy while demanding relatively less compute. This aligns with previous findings [9, 10]. *Future serving systems could improve energy efficiency for reasoning models by memory optimizations, such as KV cache compression or offloading, as the workload is constrained by memory capacity rather than raw compute power.*

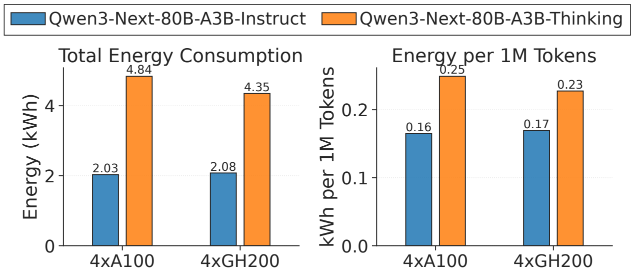


Figure 15: Power consumption of reasoning and non-reasoning models.

7.3 Limitations and Future Work

While LLMFabric successfully bridges the gap between HPC infrastructure and model serving, operating a decentralized user-space overlay at this scale has revealed limitations regarding protocol expressiveness, data locality, and service guarantees.

Protocol expressiveness and quality-of-service. As discussed earlier, the adoption of a unified, OpenAI-compatible interface simplifies client integration but abstracts away critical configuration nuances; a client cannot easily request specific trade-offs, such as lower precision for lower latency or guaranteed precision without relying on rigid naming conventions. We propose extending the serving protocol to support quality-of-service (QoS) negotiation. Future versions of LLMFabric will allow clients to specify constraints (e.g., quantization $\geq$ 4-bit, specific GPU architectures) in the request metadata, and the system will route requests to nodes that satisfy these constraints. This enhancement will empower users to make informed trade-offs between performance, cost, and model fidelity.

SLO on volatile resources. As a research-focused system, LLMFabric currently operates on a best-effort basis and does not provide strong availability guarantees. While the fail-stop design handles node termination gracefully, it does not proactively guarantee performance consistency for latency-critical applications. We aim to introduce tiered service levels. By categorizing resources based on their expected stability (e.g., differentiating between stably owned partitions and volatile scavenger queues), the scheduler can offer probabilistic guarantees. Critical workloads can be routed to stable nodes, while batch processing tasks can exploit the volatile, lower-priority capacity, maximizing cluster throughput without compromising reliability for interactive users.

Trust model in federated environments. The core value of LLMFabric, pooling heterogeneous resources via peer-to-peer routing, inherently conflicts with strict data residency and governance requirements. In scenarios where data locality is paramount (e.g., due to regulatory compliance or privacy concerns), the system must ensure that sensitive requests are not inadvertently routed to untrusted or non-compliant nodes within the mesh. Addressing this challenge requires developing more sophisticated trust models and routing policies that respect data sovereignty while still leveraging the benefits of decentralized resource pooling. Future work will explore integrating trust frameworks and more sophisticated access control mechanisms into the routing logic to balance federation benefits with sovereignty constraints.

8 Related Work

LLM serving systems. Recent works [15, 65, 68] have proposed several systems and methods for highly efficient LLM serving. AlpaServe [30] optimizes system service level objectives (SLOs) by strategically leveraging data and model parallelism. To address the resource contention between phases, DistServe [67] and Splitwise [39] disaggregate the prefill and decoding steps onto separate GPUs, whereas Sarathi-Serve [1] mitigates interference through prefill chunking and optimized request batching. In terms of memory management, vLLM [27] and SGLang [66] introduce advanced attention mechanisms, PagedAttention and RadixAttention respectively, to minimize memory fragmentation and enable efficient KV cache reuse. In the domain of scheduling and adaptation, FastServe [60] prioritizes shorter jobs via preemptive scheduling to minimize completion time, while DynamoLLM [50] adapts system configurations dynamically to optimize performance under varying workloads.

Heterogeneous LLM serving. Heterogeneous LLM serving refers to systems that utilize a mix of GPU types with varying hardware characteristics—such as differing computational and memory capacities—to optimize resource utilization and cost-efficiency. Several recent works [19, 22, 26, 40, 54] have addressed this domain. Petals [3] is the first open-source system that allows users to serve LLMs on a set of decentralized hardware. HexGen [25] introduces asymmetric partitioning and designs advanced scheduling techniques to deploy generative inference in heterogeneous and decentralized settings. ThunderServe [23] addresses the distinct computational characteristics of the prefill and decoding phases by deploying them on heterogeneous GPUs matched to their respective workload requirements. Similarly, Helix [35] formulates the optimization of heterogeneous GPU and network connections as a max-flow problem, utilizing mixed-integer linear programming to determine optimal model deployment. Collectively, these works demonstrate that the strategic utilization of heterogeneous resources significantly enhances the cost-efficiency of LLM serving. Unlike existing works on single-model serving, our service scheduler jointly addresses model heterogeneity (varying LLMs) and GPU heterogeneity (varying hardware specifications) for multi-model deployment.

Multi-tenant LLM serving. Another line of research focuses on serving multiple models concurrently to maximize resource utilization. AlpaServe [30] reveals that model parallelism can be used for statistical multiplexing of multiple models, and designs a serving system that determines efficient placing and parallelizing strategies. MuxServe [17] introduces a flexible spatial-temporal multiplexing system that identifies optimal colocation strategies for multiple LLMs. When the multiple models are fine-tuned from a shared base model, DeltaZip [61] exploits the shared weights and compresses the deltas to improve serving efficiency. If the fine-tuning is performed using parameter-efficient methods (e.g., LoRA [21]), Punica [5] and S-LoRA [48] design specialized serving systems for serving adapters efficiently. In contrast to these works, LLMFabric focuses on the challenges arising from both model and hardware heterogeneity, as well as administrative burdens in large-scale deployments, particularly on HPC environments.

9 Conclusion

We present LLMFabric, a decentralized serving system that has been operational for over 16 months. By prioritizing compatibility with existing HPC infrastructure over cloud-native paradigms, LLMFabric bridges the gap between isolated HPC cluster environments and the growing demand for large-scale sovereign AI serving infrastructure. We present the design and implementation of LLMFabric and analyze the system's real workload characteristics, which we will release in an anonymized production trace. The trace includes request information for over 15 billion tokens served across 142 models. We share our experiences and insights to inform future research on multi-tenant AI infrastructure.

References

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

[2] Arney Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Efficient llm inference via chunked prefills. *SIGOPS Oper. Syst. Rev.*, 59(1):9–16, August 2025.

[3] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 558–568, 2023.

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

[5] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13, 2024.

[6] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance, 2023.

[7] Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B. Gibbons. Slos-serve: Optimized serving of multi-slo llms, 2025.

[8] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

[9] Krishna Teja Chitty-Venkata, Siddhisanket Raskar, Bharat Kale, Farah Ferdaus, Aditya Tanikanti, Ken Raffenetti, Valerie Taylor, Murali Emani, and Venkatram Vishwanath. Llm-inference-bench: Inference benchmarking of large language models on ai accelerators. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1362–1379. IEEE, 2024.

[10] Jae-Won Chung, Jeff J Ma, Ruofan Wu, Jiachen Liu, Oh Jun Kweon, Yuxuan Xia, Zhiyu Wu, and Mosharaf Chowdhury. The ml. energy benchmark: Toward automated inference energy measurement and optimization. *arXiv preprint arXiv:2505.06371*, 2025.

[11] European Commission. Ai factories, 2025. Accessed: 2025-12-11.

[12] McKinsey & Company. The cost of compute: A $7 trillion race to scale data centers, 2025. Accessed: 2025-12-11.

[13] FastAPI Contributors. Fastapi framework, high performance, easy to learn, fast to code, ready for production, 2025. Accessed: 2025-12-10.

[14] Gin Contributors. Gin web framework, 2025. Accessed: 2025-12-01.

[15] LMDeploy Contributors. Lmdeploy: A toolkit for compressing, deploying, and serving llm. `https://github.com/InternLM/lmdeploy`, 2023.

[16] Robert Dale. Sovereign ai in 2025. *Natural Language Processing*, 31(5):1312–1321, 2025.

[17] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: Flexible spatial-temporal multiplexing for multiple llm serving. *arXiv preprint arXiv:2404.02015*, 2024.

[18] U.S. National Science Foundation. National artificial intelligence research resource pilot, 2025. Accessed: 2025-12-09.

[19] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. M\'elange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527*, 2024.

[20] Alejandro Hernández-Cano, Alexander Hägele, Allen Hao Huang, Angelika Romanou, Antoni-Joan Solergibert, Barna Pasztor, Bettina Messmer, Dhia Garbaya, Eduard Frank Ďurech, Ido Hakimi, et al. Apertus: Democratizing open and compliant llms for global language environments. *arXiv preprint arXiv:2509.14233*, 2025.

[21] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[22] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Guoliang He, Xupeng Miao, Ana Klimovic, Bin Cui, Binhang Yuan, and Eiko Yoneki. Demystifying cost-efficiency in llm serving over heterogeneous gpus. *arXiv preprint arXiv:2502.00722*, 2025.

[23] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Taiyi Wang, Bin Cui, Ana Klimovic, and Eiko Yoneki. Thunderserve: High-performance and cost-efficient llm serving in cloud environments. *arXiv preprint arXiv:2502.09334*, 2025.

[24] Youhe Jiang, Fangcheng Fu, Wanru Zhao, Stephan Rabanser, Nicholas D Lane, and Binhang Yuan. Cascadia: A cascade serving system for large language models. *arXiv preprint arXiv:2506.04203*, 2025.

[25] Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. Hexgen: Generative inference of large language model over heterogeneous environment. *arXiv preprint arXiv:2311.11514*, 2023.

[26] Youhe Jiang, Ran Yan, and Binhang Yuan. Hexgen-2: Disaggregated generative inference of llms in heterogeneous environment. *arXiv preprint arXiv:2502.07903*, 2025.

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[28] Oak Ridge National Laboratory. Summit, the next leap in leadership-class computing systems for open science., 2025. Accessed: 2025-12-11.

[29] Protocol Labs. libp2p - the peer-to-peer network stack, 2025. Accessed: 2025-12-01.

[30] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[31] Pedro Garcia Lopez, Daniel Barcelona Pons, Marcin Copik, Torsten Hoefler, Eduardo Quiñones, Maciej Malawski, Peter Pietzutch, Alberto Marti, Thomas Ohlson Timoudas, and Aleksander Slominski. Ai factories: It's time to rethink the cloud-hpc divide. *arXiv preprint arXiv:2509.12849*, 2025.

[32] Sara Vera Marjanović, Arkil Patel, Vaibhav Adlakha, Milad Aghajohari, Parishad BehnamGhader, Mehar Bhatia, Aditi Khandelwal, Austin Kraft, Benno Krojer, Xing Han Lù, et al. Deepseek-r1 thoughtology: Let's think about llm reasoning. *arXiv preprint arXiv:2504.07128*, 2025.

[33] Maxime Martinasso, Mark Klein, and Thomas Schulthess. Alps, a versatile research infrastructure. In *Proceedings of the Cray User Group*, pages 156–165. 2025.

[34] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International workshop on peer-to-peer systems*, pages 53–65. Springer, 2002.

[35] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 586–602, 2025.

[36] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *arXiv preprint arXiv:2211.13878*, 2022.

[37] World Economic Forum Muath Alduhishy. Sovereign ai: What it is, and 6 strategic pillars for achieving it., 2024. Accessed: 2025-12-11.

[38] National Institute of Advanced Industrial Science and Technology (AIST). Ai bridging cloud infrastructure (abci), 2025. Accessed: 2025-12-09.

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

[40] You Peng, Youhe Jiang, Chen Wang, and Binhang Yuan. Hexgen-text2sql: Optimizing llm inference request scheduling for agentic text-to-sql workflow. *arXiv preprint arXiv:2505.05286*, 2025.

[41] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages

155–170, Santa Clara, CA, February 2025. USENIX Association.

[42] Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, Ramachandran Ramjee, and Rodrigo Fonseca. Modserve: Modality- and stage-aware resource disaggregation for scalable multimodal model serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC 2025)*, New York, NY, USA, 2025. Association for Computing Machinery.

[43] Lukas Rauch, Raphael Schwinger, Moritz Wirth, René Heinrich, Denis Huseljic, Marek Herde, Jonas Lange, Stefan Kahl, Bernhard Sick, Sven Tomforde, and Christoph Scholz. Birdset: A large-scale dataset for audio classification in avian bioacoustics, 2024.

[44] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[45] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[46] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.

[47] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[48] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[50] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1348–1362. IEEE, 2025.

[51] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. D\'ej\avu: Kv-cache streaming for fast, fault-tolerant generative llm serving. *arXiv preprint arXiv:2403.01876*, 2024.

[52] SwissAI. Swiss ai initiative, 2025. Accessed: 2025-12-09.

[53] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. Chess: Contextual harnessing for efficient sql synthesis, 2024.

[54] Chris Tong, Youhe Jiang, Gufeng Chen, Tianyi Zhao, Sibian Lu, Wenjie Qu, Eric Yang, Lynn Ai, and Binhang Yuan. Parallax: Efficient llm inference service over decentralized environment. *arXiv preprint arXiv:2509.26182*, 2025.

[55] UncoverAlpha. The next tectonic shift in ai: Inference, 2024. Accessed: 2025-12-11.

[56] Hemish Veeraboina. Aime problem set 1983-2024, 2024.

[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. 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, page 5831–5841, New York, NY, USA, 2025. Association for Computing Machinery.

[58] Hao Wen, Xinrui Wu, Yi Sun, Feifei Zhang, Liye Chen, Jie Wang, Yunxin Liu, Yunhao Liu, Ya-Qin Zhang, and Yuanchun Li. Budgetthinker: Empowering budget-aware llm reasoning with control tokens, 2025.

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

[60] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[61] Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. Deltazip: Efficient serving of multiple full-model-tuned llms. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 110–127, 2025.

[62] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[63] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[64] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, Beidi Chen, Guangyu Sun, and Kurt Keutzer. Llm inference unveiled: Survey and roofline model insights, 2024.

[65] Li Zhang, Youhe Jiang, Guoliang He, Xin Chen, Han Lv, Qian Yao, Fangcheng Fu, and Kai Chen. Efficient mixed-precision large language model inference with turbomind. *arXiv preprint arXiv:2508.15601*, 2025.

[66] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

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

[68] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, et al. Megascale-infer: Serving mixture-of-experts at scale with disaggregated expert parallelism. *arXiv preprint arXiv:2504.02263*, 2025.

[69] ETH Zurich. The euler cluster, 2025. Accessed: 2025-12-11.