LLMServingSim 2.0: A Unified Simulator for Heterogeneous and Disaggregated LLM Serving Infrastructure

Jaehong Cho*
School of Computing
KAIST
Daejeon, South Korea
jhcho@casys.kaist.ac.kr

Hyunmin Choi*
School of Computing
KAIST
Daejeon, South Korea
hmchoi@casys.kaist.ac.kr

Guseul Heo
School of Computing
KAIST
Daejeon, South Korea
gsheo@casys.kaist.ac.kr

Jongse Park
School of Computing
KAIST
Daejeon, South Korea
jspark@casys.kaist.ac.kr

***Abstract*—Large language model (LLM) serving infrastructures are undergoing a shift toward heterogeneity and disaggregation. Modern deployments increasingly integrate diverse accelerators and near-memory processing technologies, introducing significant hardware heterogeneity, while system software increasingly separates computation, memory, and model components across distributed resources to improve scalability and efficiency. As a result, LLM serving performance is no longer determined by hardware or software choices in isolation, but by their runtime interaction through scheduling, data movement, and interconnect behavior. However, understanding these interactions remains challenging, as existing simulators lack the ability to jointly model heterogeneous hardware and disaggregated serving techniques within a unified, runtime-driven framework.**

This paper presents LLMServingSim 2.0, a unified system-level simulator designed to make runtime-driven hardware–software interactions in heterogeneous and disaggregated LLM serving infrastructures explicit and analyzable. LLMServingSim 2.0 embeds serving decisions and hardware behavior into a single runtime loop, enabling interaction-aware modeling of batching, routing, placement, offloading, memory management, and power consumption. The simulator supports extensible integration of emerging accelerators and memory systems through profile-based modeling, while capturing dynamic serving behavior and system-level effects. We validate LLMServingSim 2.0 against real serving deployments, showing that it reproduces key performance, memory, and power metrics with an average error of 0.95%, while maintaining practical simulation times of around 10 minutes even for complex system configurations. These results demonstrate that LLMServingSim 2.0 provides a practical bridge between hardware innovation and serving-system design, enabling systematic exploration and co-design for next-generation LLM serving infrastructures. Our simulator is available at https://github.com/casys-kaist/LLMServingSim.

***Index Terms*—Large language model (LLM), Inference serving, Heterogeneous hardware, Simulation infrastructure**

I. Introduction

Large language model (LLM) serving infrastructures are becoming increasingly *heterogeneous*, moving beyond the homogeneous, GPU-only deployments in most existing systems. While NVIDIA GPUs remain central, modern deployments increasingly integrate diverse hardware, including hyperscalers' accelerators such as Google TPU [1] and Amazon Inferentia [2], as well as emerging domain-specific NPUs [3]–[8]. In parallel, memory-centric architectures, including processing-in-memory (PIM) and processing-near-memory (PNM) technologies actively developed by memory vendors such as Samsung [9], [10] and SK hynix [11], [12], further expand the heterogeneity of the serving substrate. These trends indicate a shift in which LLM serving performance is shaped by the coordinated use of diverse compute and memory devices, making heterogeneity an inherent characteristic of modern LLM serving infrastructure.

*These authors contributed equally to this work.

TABLE I
Comparison of LLM serving simulators

Simulator	Dissagg.			Parallelism			Modeling			
	PD	AF	HT	PP/TP	DP	EP	PA	PC	EO	PM
LLMServingSim [25]	×	✓	×	✓	×	×	✓	×	×	×
Vidur [26]	×	×	×	✓	✓	×	△	×	×	×
APEX [27]	×	×	×	✓	✓	✓	×	×	×	✓
TokenSim [28]	✓	×	×	✓	△	×	✓	△	×	×
Ours	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

PD: Prefill/Decode Disaggregation, AF: Attention/FFN Disaggregation, HT: Heterogeneous System, PP/TP: Pipeline/Tensor Parallelism, DP: Data Parallelism, EP: Expert Parallelism, PA: PagedAttention, PC: Prefix Caching, EO: Expert Offloading, PM: Power Modeling. ✓: fully supported, ×: not supported, △: limited or partial support.

Alongside this shift, *disaggregation* has emerged as a fundamental structural pattern in modern LLM serving systems. As deployments scale beyond a single server or a device type, computation, memory, and model components are increasingly separated across distributed resources to improve efficiency and flexibility. Modern serving techniques reflect this co-evolution by explicitly disaggregating execution through mechanisms such as prefill–decode separation [13], [14], mixture-of-experts with expert parallelism and offloading [15]–[17], and memory-level disaggregation enabled by prefix caching [18]–[20] and remote memory pools [21]–[24]. In practice, disaggregation and heterogeneous resource utilization often arise together as complementary design choices for optimizing performance and efficiency, jointly shaping the dominant architectural direction for scaling LLM serving.

In this landscape, a key challenge for researchers designing new hardware and system architectures is to understand how heterogeneous devices and disaggregated system structures interact in practice. Performance, efficiency, and scalability are no longer determined by hardware or software choices in isolation, but by how these choices interact at runtime, including their impact on scheduling decisions, memory behavior, and interconnect contention. Gaining such insights directly from deployed systems is costly and often impractical, requiring substantial engineering effort and large-scale infrastructure even to explore high-level design trade-offs. Consequently, simulation tools play a critical role in enabling early-stage exploration of hardware–software co-design for LLM serving. However, as shown in Table I, existing simulators fall short of this need, as they do not adequately model the runtime interactions between heterogeneous hardware and disaggregated serving techniques within a unified framework.

2766-0486/26/$31.00 ©2026 IEEE
DOI 10.1109/ISPASS69572.2026.00012

To address this gap, we present LLMServingSim 2.0, a unified simulator designed to make hardware–software interactions in heterogeneous and disaggregated LLM serving systems explicit and analyzable. LLMServingSim 2.0 embeds serving decisions and hardware behavior within a single, runtime-driven simulation loop, allowing batching, routing, placement, and offloading decisions to adapt dynamically to system conditions. By modeling how these decisions evolve and propagate over time, the simulator enables direct exploration of performance and efficiency behaviors that arise from hardware–software interaction, rather than from static configurations. As summarized in Table I, LLMServingSim 2.0 brings together capabilities that are only partially supported in prior simulators, establishing a unified framework for interaction-aware evaluation of modern LLM serving infrastructures.

We realize these capabilities through a set of modeling contributions that together form the unified framework:

- **Interaction-awareness.** LLMServingSim 2.0 explicitly models the runtime interactions between serving software decisions and heterogeneous hardware behavior, capturing how batching, routing, placement, caching, and offloading adapt to evolving system state and how their effects propagate over time. By treating these interactions as first-class modeling targets, the simulator enables analysis of performance behaviors that arise from feedback between software policies and hardware conditions.
- **Unified modeling of heterogeneity and disaggregation.** The simulator provides a unified representation of heterogeneous accelerators, multi-tier memory systems, and disaggregated serving architectures, enabling coherent evaluation of system behaviors that emerge only from their combined interaction. This unified view allows researchers to study how disaggregated serving techniques behave under different hardware compositions without isolating hardware and software effects.
- **Runtime-driven serving dynamics.** By embedding serving decisions into a runtime-driven simulation loop, LLMServingSim 2.0 allows system performance to emerge from dynamic request flows, resource contention, and interconnect effects, rather than from static or a priori configurations. This approach captures temporal effects such as queueing, contention amplification, and phase-dependent behavior that are difficult to observe with static models.
- **Extensibility to emerging hardware.** Through profile-based operator modeling, LLMServingSim 2.0 supports low-effort integration of new accelerators and memory technologies, enabling evaluation of future hardware designs without restructuring the serving model. This design facilitates rapid exploration of new hardware–software combinations as LLM serving infrastructures continue to evolve.
- **Power-aware modeling.** LLMServingSim 2.0 incorporates power modeling into the simulation framework, enabling joint evaluation of performance and energy-related trade-offs in LLM serving systems. By associating compute, memory, and data-movement activities with power characteristics, the simulator allows researchers to analyze how serving strategies, hardware heterogeneity, and disaggregation choices impact power consumption and energy efficiency alongside performance.

We evaluate LLMServingSim 2.0 by validating its accuracy, efficiency, and practical usefulness as a system-level simulation tool for LLM serving infrastructure. Specifically, we compare simulated serving performance against real deployments, showing that LLMServingSim 2.0 reproduces key metrics such as throughput, latency breakdowns (e.g., TTFT and TPOT), memory usage, and power consumption with an average error of 0.95% across representative workloads and hardware platforms. We further demonstrate that the simulator enables efficient exploration of heterogeneous and disaggregated serving configurations, while maintaining practical simulation times on the order of minutes even for complex system setups, without sacrificing architectural fidelity. Together, these results show that LLMServingSim 2.0 serves as a practical bridge between hardware innovation and serving-system design, enabling systematic co-design and early-stage exploration for the evolving LLM serving infrastructure ecosystem.

II. Background

A. LLM Inference Serving: Workflow and System Structure

Modern LLM inference follows a two-phase execution workflow consisting of a compute-intensive *prefill* stage and a memory-intensive *decode* stage [29]. Prefill stage processes the input sequence using dense matrix multiplications, while the decode stage generates tokens autoregressively with repeated key-value (KV) cache accesses. As such, prefill stresses compute throughput, whereas decode is primarily constrained by memory bandwidth, capacity, and KV cache locality.

In practical deployments, data-center scale serving systems concurrently operate tens to hundreds of model instances [30]–[33]. Incoming requests induce dynamic batching and queuing, which result in time-varying compute and memory utilization. To meet latency and throughput requirements under such variability, modern serving frameworks employ diverse techniques, including tensor, pipeline, and data parallelism [34], expert parallelism for mixture of experts (MoE) models [15], [17], [35], prefix caching for KV cache reuse [20], [36]–[40], and prefill-decode (PD) disaggregation across separate clusters [13], [14], [40], [41]. These techniques reshape execution order, memory access patterns, communication behavior, and device utilization during inference serving.

B. Heterogeneous Accelerators and Memory Hierarchies

LLM serving today is built on increasingly heterogeneous hardware platforms. A single cluster may host GPUs of different generations [14], NPUs and TPUs [1], [42]–[44] specialized for matrix operations, memory-centric accelerators such as Processing-in-Memory (PIM) [45]–[48], and systems augmented with Compute Express Link (CXL)-based devices [23], [49]–[51]. These accelerators differ substantially in compute capability, on-device memory bandwidth, memory capacity, interconnect throughput, and power characteristics, resulting in a highly non-uniform compute substrate.

Memory hierarchy plays a central role in LLM inference

performance. KV cache accesses span multiple memory tiers, including accelerator HBM, host DRAM, storage devices, and large CXL memory pools, each with distinct latency and bandwidth characteristics [36], [40], [52], [53]. During decode, the efficiency of KV cache placement, reuse, and migration directly influences latency and throughput. For MoE models, expert weights may be partitioned across devices or offloaded to memory-rich units, further shaping the pattern of operator execution and inter-device communication [15], [17], [35].

Together, these trends illustrate that LLM serving operates over a diverse and multi-tiered compute-memory stack. Execution behavior, data movement, and device heterogeneity interact in complex ways, motivating the need for modeling tools that capture the characteristics of modern LLM serving.

III. Motivation

A. Modeling Complex Runtime Dynamics in LLM Serving

LLM serving performance is shaped not by isolated operator execution but by the evolution of runtime dynamics driven by incoming requests. As requests arrive, queues form, batch sizes fluctuate, and schedulers adapt placement decisions based on device and memory pressure. KV cache blocks are allocated, reused, or migrated across tiers; prefix caching introduces hit and miss patterns; and MoE models incur token-dependent expert routing. These tightly coupled events determine time-to-first-token (TTFT), time-per-output-token (TPOT), end-to-end latency distribution, and throughput. Static or operator-only performance models cannot capture such temporal effects, including shifting batch composition, locality-dependent KV cache behavior, heterogeneous-device cooperation, or routing imbalance. Accurate evaluation requires a simulator capable of representing these interactions as they unfold over time.

B. Limitations of Existing LLM Serving Simulators

Existing LLM simulation tools fall into two broad categories, each modeling different aspects of the serving stack while leaving key serving behaviors insufficiently captured. Hardware-centric simulators, such as LLMCompass [54] and ADOR [55], enable detailed modeling of low-level accelerator behavior and communication. However, their focus on static execution prevents them from capturing dynamic serving-time behaviors, such as request arrivals, batching evolution, and KV cache reuse, which are critical to end-to-end serving performance. System-level simulators, including Vidur [26] and APEX [27], facilitate exploration of optimal execution configurations and scheduling policies at the serving layer. However, their modeling of memory management and heterogeneous deployments is limited, particularly for features that form the core of recent serving systems such as prefix caching and PD disaggregation. TokenSim [28] takes a step toward modeling serving dynamics, but its simplified memory abstraction restricts fidelity when studying KV placement, bandwidth contention, and migration. Finally, LLMServingSim [25] focuses on collecting runtime statistics in a single-instance setting, but it is not designed to study multi-instance or heterogeneous setups, nor serving features such as MoE behavior, prefix caching, or PD disaggregation.

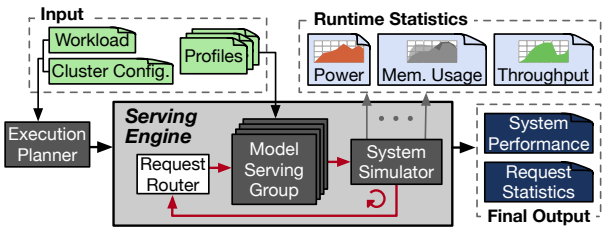


Fig. 1. Overview of LLMServingSim 2.0.

C. Need for a Unified Simulator

Modern LLM serving systems operate at the intersection of heterogeneous accelerators, multi-tier memory, multiple parallelism schemes (including MoE), prefix caching, and multi-instance routing with disaggregation. These components interact in nontrivial ways, producing behaviors that cannot be captured by partial or layer-specific models. Supporting hardware-software co-design, policy evaluation, and architectural exploration therefore requires a unified simulator that integrates hardware fidelity, memory hierarchy dynamics, serving techniques, and fully dynamic request flows. This need motivates the development of LLMServingSim 2.0, which provides a cohesive, extensible framework for modeling end-to-end LLM serving behavior on heterogeneous systems.

IV. LLMServingSim 2.0

A. Overview

Inputs. As shown in Fig. 1, LLMServingSim 2.0 takes three input specifications: (1) workload configuration, (2) cluster configuration, and (3) hardware performance profiles. The workload configuration describes the LLM and request patterns, such as arrival rates and per-request execution traces. The cluster configuration defines the deployment environment, including node type and count, CPU settings, memory capacity and bandwidth, and device placement. It also defines serving policies, including request routing, parallelism strategies, KV cache eviction, and compute and memory offloading decisions.

Profile-based modeling. To enable fast and accurate evaluation at scale, LLMServingSim 2.0 employs profile-based operator performance modeling. We develop an *Operator-level Profiler* built on a PyTorch/HuggingFace-based profiling API, which requires only a single device and minimal code changes, allowing users to collect device profiles without modifying model implementations. A one-time profiling pass measures operator-level latency and power using only a single decode block per model-device pair, typically completing within 2.1 hours (e.g., Llama 3.1-70B on NVIDIA H100). Collected profiles can be reused across experiments without re-running hardware profiling. In addition to real-hardware profiling, LLMServingSim 2.0 can ingest operator-level profiles from external hardware simulators, enabling the evaluation of future accelerators such as PIM without requiring physical hardware.

Serving workflow. Simulation begins with a one-time initialization step. During initialization, the *Execution Planner* constructs the *Serving Engine* by instantiating a *Model Serving Group* (MSG) per model, assigning devices and serving policies, and configuring memory, power, and system topol-

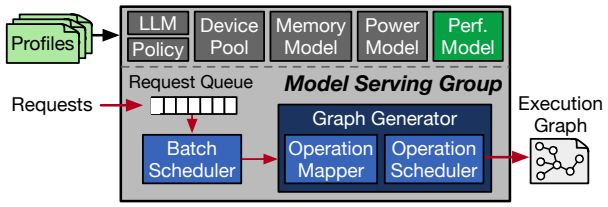


Fig. 2. Model Serving Group (MSG) architecture. The MSG includes a customizable device pool enabling heterogeneous hardware composition and supports batching, operation mapping, and scheduling for LLM execution.

ogy within the *System Simulator*. After this setup, execution proceeds in a runtime loop. The *Request Router* dispatches incoming requests to the appropriate MSG, which generates an execution graph for each request batch. The System Simulator then evaluates the graph, accounting for communication, synchronization, and multi-tier memory access. The loop repeats until all requests are complete, producing both online runtime statistics and final serving performance.

Outputs. During execution, LLMServingSim 2.0 reports system-level metrics, such as memory usage, energy consumption, and throughput. After completion, it reports per-request serving metrics, including TTFT, TPOT, queueing delay, and end-to-end latency. Together, these outputs offer a comprehensive view of LLM serving and enable quantitative comparisons across hardware configurations and policy choices.

B. Execution Planner

The Execution Planner takes the workload and cluster configurations and performs a one-time initialization of the Serving Engine. It configures the Request Router to generate and route runtime requests to the appropriate MSGs. The planner then creates an MSG for each model, assigns devices to form a customizable device pool, and installs serving policies, including parallelism strategies, compute and memory offloading, KV cache management, and memory sharing. These flexible policies enable systematic exploration of diverse serving deployments, including heterogeneous configurations.

Each MSG is populated with operator-level performance profiles and serving policies, enabling batching, mapping, and scheduling at runtime, as illustrated in Fig. 2. In parallel, the planner initializes the System Simulator with cluster-level timing, network, memory, and topology configurations for system-wide performance evaluation. After initialization, execution proceeds entirely within the Serving Engine, where the Request Router, MSGs, and the System Simulator interact iteratively until all requests complete.

C. Model Serving Group

A Model Serving Group (MSG) is a logical execution unit responsible for serving one LLM instance. As shown in Fig. 2, each MSG maintains a device pool containing one or more accelerators, which users may configure to reflect the desired serving deployment. The pool serves as the execution substrate for operator mapping and may comprise a mix of accelerators and memory devices, such as GPUs, NPUs, CXL-attached devices, and PIM-equipped memory channels. All execution decisions reflect the configured LLM, serving policies, and the simulator's power, memory, and performance models.

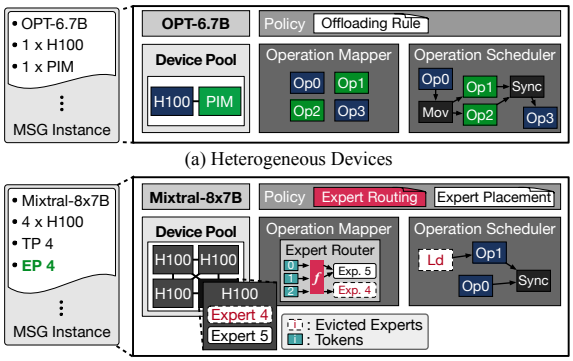
Request queue. The MSG receives requests from the router and tracks them until completion. Each request progresses through prefill and decode while accumulating statistics such as queueing delay, TTFT, TPOT, and end-to-end latency.

Batch scheduler. The batch scheduler selects pending requests from the queue and forms an executable batch. It evaluates system and device memory capacity, the KV cache footprint, and the configured maximum batch size. The memory model is consulted to determine whether the candidate batch fits within available resources, including prefix-cache residency and eviction cost. If the batch satisfies these constraints, it is forwarded to the graph generator, where execution begins at the operation mapper.

Memory model. The memory model governs KV cache management and device memory usage throughout inference serving. During batch scheduling, KV cache eviction and promotion decisions are derived based on per-device memory capacity and cache residency, and the corresponding load and store overheads are injected into the execution graph. Prefix KV cache placement across multiple memory tiers is explicitly modeled, including device memory, host memory, CXL-attached memory, and storage, while accounting for caching policies, capacity constraints, and data movement costs. Together, these mechanisms enable accurate simulation of memory placement and movement across tiers, as well as contention for memory resources during inference serving.

Power model. The power model tracks the instantaneous power consumption of each MSG while simultaneously accounting for the total system-level energy consumption. For each MSG, the power model is decomposed into seven major components that account for the majority of energy consumption in modern servers: accelerators, CPUs, DRAM, interconnect links (including switches), Network Interface Cards (NICs), storage devices, and remaining system components such as the motherboard and cooling infrastructure. Accelerator power is modeled using a three-state model comprising idle, active, and standby states to capture utilization-dependent behavior. DRAM and interconnect links consume energy proportional to the volume of data transferred, while the remaining components are modeled with constant power. This design enables tracking runtime power usage across different MSG configurations and workloads, and quantifying how serving policies and system configurations influence energy consumption at both the MSG and system levels.

Operation mapper. The operation mapper assigns operators to devices within an MSG based on the configured parallelism strategies and optional compute and memory offloading rules. When multiple device types are available, operator placement further incorporates fine-grained operation-level offloading decisions, such as executing attention on PIM or offloading experts and KV caches to specific memory devices. For each assignment, the simulator attaches latency and power estimates using the operator profile. Once mapping is complete, the operator set is forwarded to the operation scheduler for dependency resolution and directed acyclic graph (DAG) construction.



(a) Heterogeneous Devices

(b) Mixture of Experts

Fig. 3. Model serving group-level usage example.

Operation scheduler. The operation scheduler constructs an execution DAG that encodes data dependencies, ordering constraints, parallelism strategies, and the required communication and memory operations. The final output is an execution graph, which is passed to the System Simulator as input.

D. System Simulator

The System Simulator executes the operator graphs generated by each MSG and evaluates end-to-end execution at the cluster scale. Each node in the execution graph represents an operator annotated with device type, latency, memory, and communication statistics, and power information, while edges encode data dependencies that determine scheduling and parallelism. During runtime, the simulator models synchronization overhead, network contention, inter-device communication, and memory access latency based on cluster topology and configuration.

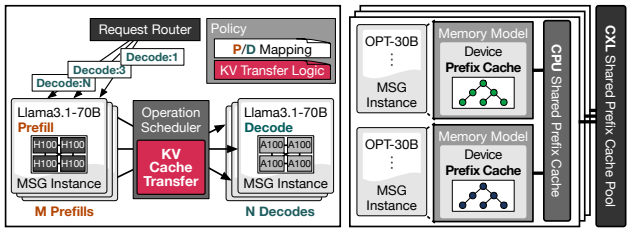
To support cluster-scale timing and memory evaluation, LLMServingSim 2.0 builds on a modified version of ASTRA-sim [56] and Chakra [57]. Those existing frameworks, originally designed for simple and repetitive training patterns, are insufficient for modeling dynamic LLM inference serving. We therefore extend both frameworks to support heterogeneous compute fabrics and operator-driven execution graphs that capture the dynamics of LLM inference and support inference-specific parallelism, such as expert parallelism.

In addition, LLMServingSim 2.0 integrates a refined memory model to capture bandwidth contention, KV movement, and memory sharing with higher fidelity. To enable such modeling, we extend the memory hierarchy to include device memory, host memory, storage, and CXL-attached memory, and explicitly model memory sharing to capture cluster-wide effects. Furthermore, we add PIM operations beyond memory load and store primitives, enabling system-level simulation of PIM-accelerated execution. As a result, LLMServingSim 2.0 simulates LLM inference serving by jointly modeling operator-driven execution, inference dynamics, and heterogeneous memory systems within the Serving Engine loop.

V. Serving Techniques and Use-Case Modeling

A. Model Serving Group-Level Techniques

Heterogeneous devices and operator-granular offloading. LLMServingSim 2.0 allows a single MSG to combine heterogeneous devices and execute an LLM with device-specialized operator placement. During initialization, the Execution Planner constructs the MSG device pool and installs serving policies that specify how different devices are utilized. At runtime, the MSG's operation mapper assigns each operator to an appropriate device at operator granularity based on these policies. As illustrated in Fig. 3(a), when the device pool includes both GPUs and PIM modules and attention offloading is enabled, attention operators are mapped to PIM, while the remaining operators execute on GPUs.



(a) Prefill-Decode Disaggregation (b) Prefix Caching and Multi-MSG Sharing

Fig. 4. System-level usage example.

The operation scheduler then incorporates the required data-movement and synchronization operations into the execution graph. For attention-offloading, this involves transferring intermediate activations and KV caches from GPUs to PIM before execution and returning the results to GPUs afterward. These operations are automatically inserted, executed by the System Simulator, and included in latency modeling. Through this design, LLMServingSim 2.0 supports flexible operator-granular offloading across heterogeneous devices via simple policy configuration, enabling efficient exploration of heterogeneous serving systems.

MoE via expert routing, parallelism, and offloading. For MoE models, LLMServingSim 2.0 similarly supports operator-level modeling of the full MoE execution flow within a single MSG. MoE behavior is controlled by serving policies installed by the Execution Planner, including expert placement across devices, expert offloading rules, and routing strategies. Each MSG instantiates a configurable *Expert Router* that emulates gating behavior and determines expert assignment on a per-token basis. The router supports a variety of routing schemes, such as random selection, round-robin routing, proportional-load balancing, as well as user-defined policies.

During serving, as shown in Fig. 3(b), tokens are routed to experts at each MoE layer according to the routing policy. Depending on the configuration, experts may reside on different devices or be temporarily evicted to host memory. The operation scheduler automatically translates expert loading and cross-device communication into an execution graph, without requiring explicit user specification. This execution graph is evaluated by the System Simulator, which accounts for latency from expert loading, token routing, and all-to-all communication in expert parallelism. Through this mechanism, LLMServingSim 2.0 enables exploration of MoE execution dynamics under diverse routing and offloading scenarios.

B. System-Level Serving Techniques

Prefill-decode disaggregation across serving groups. LLMServingSim 2.0 supports PD disaggregated deployments by allowing the Execution Planner to instantiate multiple prefill and

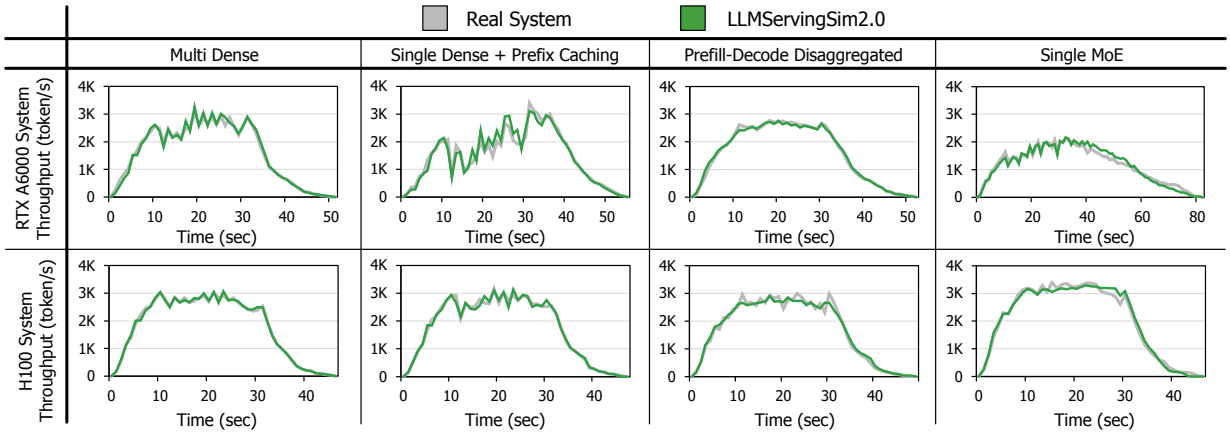


Fig. 5. Comparison of throughput over time between real GPU systems (RTX A6000, H100) and LLMServingSim 2.0 using the vLLM framework.

decode MSGs for the same model. These MSGs coordinate to serve the same model but can be configured heterogeneously, with different hardware types, memory hierarchies, and interconnects. Based on serving policies, prefill and decode MSGs can be mapped with arbitrary M:N ratios.

As shown in Fig. 4(a), at runtime, the Request Router dispatches incoming requests to a prefill MSG along with the associated decode MSG information. The prefill MSG operation scheduler then automatically inserts layer-wise KV cache transfer operations to the decode MSG, constructing the corresponding execution graph. These KV movements are handled by the memory model and evaluated by the System Simulator, which computes inter-node communication latency according to the configured network topology and bandwidth. Through this design, LLMServingSim 2.0 enables exploration of PD disaggregated systems with heterogeneous hardware configurations and flexible prefill-decode mapping.

Prefix caching and multi-MSG sharing. LLMServingSim 2.0 supports prefix caching across multiple memory tiers. Based on the configured caching policy, the Execution Planner instantiates radix-tree-based prefix caches at the appropriate tiers when constructing MSGs. As shown in Fig. 4(b), when prefix caching is limited to devices, each MSG maintains its own prefix cache within its memory model. When CPU host memory is used as a shared second-tier cache, MSGs on the same node retain local device-level prefix caches while additionally sharing a common CPU-resident prefix cache. When a CXL memory pool is enabled, all MSGs access a single globally shared prefix cache across the system.

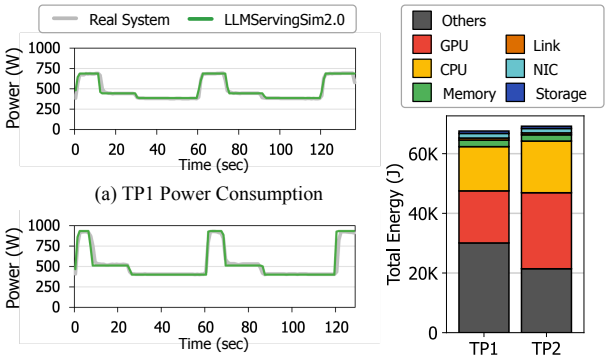
During simulation, as requests pass through the MSG operation mapper, prefix hits reduce the effective execution latency. If the required KV cache is not present on the device, the operation scheduler automatically inserts layer-wise KV transfer operations from the appropriate memory tier, which are executed and evaluated by the System Simulator. With this multi-tier shared prefix cache design, LLMServingSim 2.0 enables systematic exploration of diverse prefix caching policies and supports research on multi-tier caching behavior in large-scale LLM serving systems.

VI. Methodology

System baseline. We evaluate LLMServingSim 2.0 on three representative serving platforms: (1) an on-premise GPU server with four NVIDIA RTX A6000 GPUs and an Intel Xeon Gold 6326 CPU, (2) a cloud-based system with eight NVIDIA H100-SXM-80GB GPUs, and (3) a TPU-v6e-1 instance on Google Cloud. Across all platforms, we use vLLM [58] as the serving framework. For model coverage, we evaluate both dense and MoE models: Llama 3.1-8B and Phi-mini MoE on the RTX A6000 and TPU systems, and Llama 3.1-70B and Mixtral 8×7B on the H100 system with a tensor parallelism degree of four. Unless otherwise specified, all workloads used in evaluations are generated by sampling 300 requests from ShareGPT [59], with request arrivals synthesized using a Poisson process at a rate of 10 requests per second.

LLMServingSim 2.0 configuration. To evaluate LLMServingSim 2.0, we integrate GPU and TPU backends by extracting performance models through Operator-level Profiler. For GPUs, we profile the target LLMs on RTX A6000 and H100 systems and configure the simulator with matching device specifications, including memory capacity, memory bandwidth, and interconnect characteristics. Specifically, the simulator is parameterized with 40 GB and 80 GB of device memory, 936 GB/s and 3.35 TB/s memory bandwidth, and PCIe 4.0×16 and NVLink interconnects for RTX A6000 and H100, respectively. To demonstrate extensibility beyond GPUs, we extend the profiler to a TPU-v6e-1 instance and configure the simulator with corresponding specifications, including 32 GB of memory capacity, 1.6 TB/s memory bandwidth, and an interconnect bandwidth of 800 GB/s.

Simulator baseline. We further compare the simulation accuracy and execution time of LLMServingSim 2.0 against existing LLM serving system simulators, including Vidur [26], APEX [27], TokenSim [28], and LLMServingSim [25]. Because these baselines do not support all serving features modeled in LLMServingSim 2.0, we limit the comparison to configurations and workloads that are executable by at least one baseline simulator.

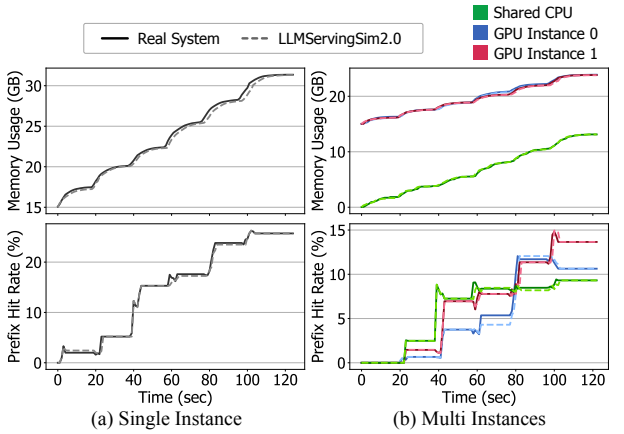


(a) TP1 Power Consumption

(b) TP2 Power Consumption

(c) Energy Breakdown

Fig. 6. Comparison of power consumption and energy breakdown between the real RTX A6000 system and LLMServingSim 2.0.



(a) Single Instance

(b) Multi Instances

Fig. 7. Comparison of memory usage and prefix hit rate between the real RTX A6000 system and LLMServingSim 2.0.

VII. Evaluation

A. Validation with Real Serving System

Performance. We evaluate LLMServingSim 2.0 by comparing its results against measurements from real GPU systems running vLLM, demonstrating consistent accuracy across different GPU platforms. Fig. 5 shows the time-series throughput of the real system and LLMServingSim 2.0 under various serving configurations, including multi-dense model serving, prefix caching, PD disaggregation, and MoE model serving.

Across both RTX A6000 and H100 systems, LLMServingSim 2.0 closely tracks the temporal throughput patterns observed in the real system. On a time-series basis, the average throughput error is 5.14% on RTX A6000 and 3.29% on H100, reflecting runtime variability from dynamic batching, request arrivals, and execution phase transitions. This consistency holds across different serving modes and model types, despite their distinct execution and memory behaviors. When performance metrics are aggregated over the full execution, including both throughput and latency, the average error drops to 0.99% and 1.54% for RTX A6000 and H100, respectively. These results demonstrate that LLMServingSim 2.0 captures both dynamic behavior and end-to-end performance trends across diverse serving configurations and hardware platforms.

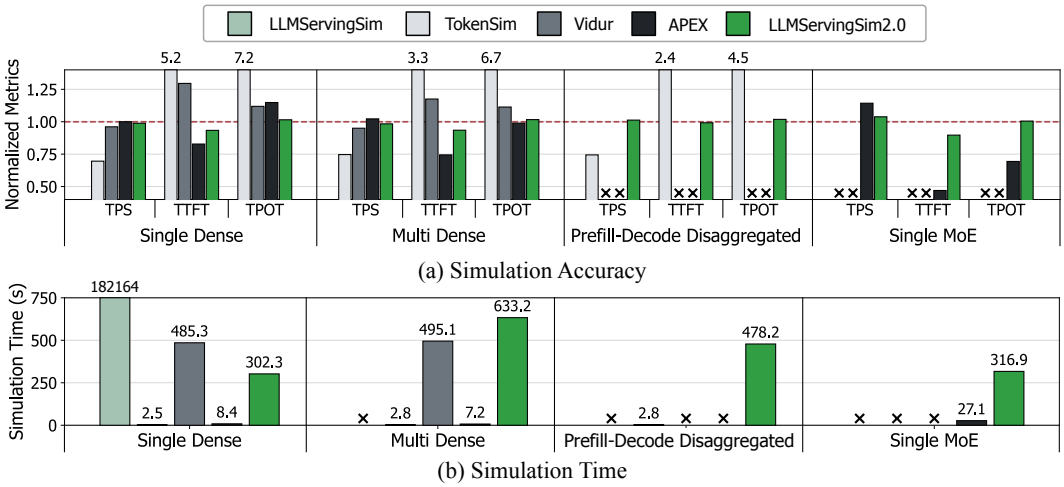
Power consumption. Fig. 6(a) and (b) compare the real-time power consumption of an RTX A6000 system with that predicted by LLMServingSim 2.0 under tensor parallelism degrees 1 and 2. In this experiment, we generate each request pulse by parsing 10 requests from the ShareGPT dataset and issuing them three times at 60-second intervals. Idle gaps are intentionally inserted between pulses to exercise the three-state accelerator power model in LLMServingSim 2.0, which alternates among idle, active, and standby states.

As shown in Fig. 6(a) and (b), LLMServingSim 2.0 closely matches the measured power consumption of the RTX A6000 system under both tensor parallelism configurations, with an average error of 1.34% in total energy consumption. Three distinct power pulses are observed, corresponding to transitions among active, standby, and idle states. Higher tensor parallelism activates more GPUs, resulting in higher peak power, while shorter execution time leads to narrower power peaks. These results confirm that LLMServingSim 2.0's three-state power model and other power components accurately capture the runtime power dynamics of the real system.

Fig. 6(c) presents the energy breakdown across seven power components. The others category is relatively large because unused GPUs in the RTX A6000 server, which contains four GPUs in total, are modeled as constant power consumers. Among the remaining components, accelerators dominate energy consumption, followed by CPUs and memory, consistent with a real system. This accurate power modeling and fine-grained energy breakdown enable LLMServingSim 2.0 to predict power consumption and evaluate energy efficiency under diverse hardware, workloads, and serving policies.

Memory usage. Fig. 7 compares the temporal trend of memory usage and prefix cache hit rate between RTX A6000 systems and LLMServingSim 2.0 under single-instance and multi-instance serving configurations, using the same model and parallelism configuration. For GPU-based prefix caching, we use the native vLLM prefix caching mechanism with a block size of 16. For CPU-based prefix caching and centralized KV cache sharing, we integrate the LMCache [39] framework with vLLM, configured with a block size of 256. Workloads are sampled from ShareGPT to capture request lengths and prefix reuse patterns. Request arrivals are synthesized with alternating bursty and idle periods to reproduce the temporal fluctuations in memory usage and prefix cache hit rates observed in real serving systems.

Fig. 7(a) shows results for a single-instance setup. In this setting, LLMServingSim 2.0 closely matches the real system in both memory usage and prefix cache hit rate, with an average error of 0.93%, capturing the step-wise growth in device memory footprint and the corresponding increase in prefix reuse as requests accumulate. Fig. 7(b) presents a multi-instance setting with centralized CPU-based prefix cache sharing, where two instances maintain separate device memory pools while accessing a shared CPU prefix cache pool. In this setting, the real system exhibits dynamic memory usage across instances and a higher aggregate prefix cache hit rate enabled by cross-instance reuse. LLMServingSim 2.0 reproduces these



(a) Simulation Accuracy

(b) Simulation Time

Fig. 8. Comparison with existing LLM serving simulators in terms of throughput (TPS), TTFT, TPOT, and simulation time. (a) Normalized to vLLM results.

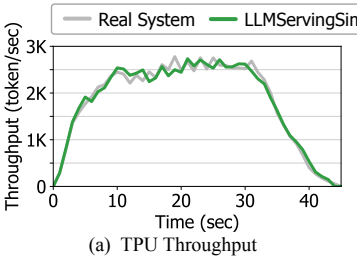
behaviors with an average error of 0.41%, accurately capturing per-instance memory growth and synchronized increases in prefix cache hit rate over time.

B. Comparison with Other Simulators

Fig. 8 compares LLMServingSim 2.0 with prior LLM serving simulators in terms of simulation accuracy and time across diverse serving configurations. We evaluate Vidur [26], APEX [27], TokenSim [28], and the original LLMServingSim [25], with all performance metrics normalized to a real GPU system running vLLM. Configurations unsupported by baseline simulators are marked as unavailable.

Fig. 8(a) reports normalized throughput (TPS), TTFT, and TPOT. Across single- and multi-instance dense serving configurations, LLMServingSim 2.0 achieves high accuracy across all metrics, with an average error of 2.43%. Prior simulators tend to be accurate only for specific metrics, and show larger deviations on others. Under more complex configurations, including PD disaggregation and MoE serving, several baselines fail to execute or rely on simplified abstractions, resulting in large errors. In contrast, LLMServingSim 2.0 remains accurate across all supported metrics, achieving an average error of 1.81%, demonstrating its ability to jointly model serving dynamics and memory behavior under complex serving configurations.

Fig. 8(b) reports simulation time. Compared to lightweight simulators such as Vidur and TokenSim, LLMServingSim 2.0 incurs higher simulation overhead due to its detailed modeling of serving dynamics and memory behavior. This overhead is an inherent trade-off of LLMServingSim 2.0's design goal: supporting heterogeneous accelerator types and hardware combinations while accurately capturing system-level interactions. Although this leads to longer simulation time than GPU-centric baselines, it enables simulation of serving behavior on heterogeneous platforms beyond their scope. Moreover, compared to the original LLMServingSim, LLMServingSim 2.0 substantially reduces simulation time while expanding the supported design space, striking a practical balance between modeling fidelity and simulation efficiency.



(a) TPU Throughput

Error Rate (%)	
TPS	0.21795
TPOT	0.23349
ITL	0.50865
Geomean	**0.29581**

(b) Error Rate

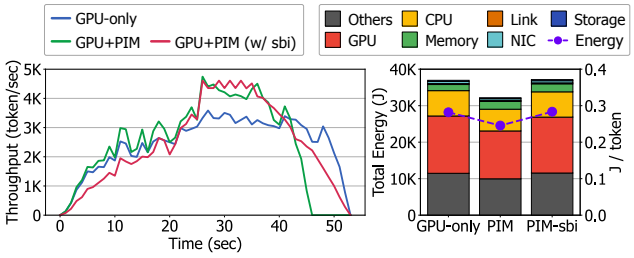
Fig. 9. Comparison of throughput over time and latency metrics between real TPU system and LLMServingSim 2.0 using vLLM framework.

C. Case Study: Emerging Hardware

LLMServingSim 2.0 supports emerging hardware platforms, including TPU and PIM. As long as per-operator latency models are available through specification or profiling, these platforms can be incorporated into the simulation framework. We demonstrate this capability through case studies.

TPU. To demonstrate LLMServingSim 2.0 's ability to integrate emerging accelerators beyond GPUs, we conduct a case study using a TPU-v6e-1 instance on Google Cloud. We deploy a real TPU serving system using the TPU-enabled vLLM framework and extract an operator-level performance model for TPU-v6e-1 via a TPU-specific profiler. As the current vLLM-TPU framework stably supports only single-instance dense serving, our validation is limited to this configuration. We use Llama 3.1-8B with a tensor parallelism degree of one and reuse the same workload trace as in the GPU experiments. The simulator is configured to match the TPU-v6e-1 memory capacity and bandwidth, enabling accurate reproduction.

Fig. 9 compares the time-series throughput of the real TPU system and LLMServingSim 2.0 under the same workload. Despite architectural differences between GPUs and TPUs, LLMServingSim 2.0 closely tracks the real system, with an average per-timestep throughput error of 3.95%. When performance is aggregated over the full execution, the average error goes below 0.3% across all evaluated metrics. These results show that LLMServingSim 2.0 can effectively model LLM serving behavior on non-GPU accelerators when appropriate



(a) Token Generation Throughput (b) Energy Breakdown

Fig. 10. Performance comparison between a GPU-only system and a GPU+PIM system, including sub-batch interleaving (SBI).

operator-level performance models are available. Moreover, LLMServingSim 2.0 enables hypothetical analysis of serving techniques not yet supported by current TPU frameworks, such as PD disaggregation and prefix caching, allowing early exploration of their system-level impact.

Processing-in-Memory. We conduct a case study comparing a GPU-only system and GPU+PIM systems, including a variant with sub-batch interleaving (SBI) proposed in NeuPIMs [45]. In the GPU-only setup, Llama 3.1-8B runs on a single RTX A6000 GPU. In the GPU+PIM setup, main memory is replaced with a PIM system composed of 256 channels, each equipped with 1 GB of HBM2 at 2000 MT/s. In the PIM setup, requests are distributed across PIM channels. To reflect this behavior, we use a workload of 256 requests, matching the number of PIM channels, with input and output of 128 and 512 tokens.

Fig. 10(a) compares the throughput of GPU-only and GPU+PIM systems over time. During the prefill-dominated period (0–25 s), performance gains are limited, as PIM primarily accelerates the memory-intensive decode phase. After prefill completes, GPU+PIM achieves 1.43× higher throughput than the GPU-only system, demonstrating the effectiveness of PIM for decode workloads. In contrast, GPU+PIM with SBI shows performance comparable to the GPU-only system. Although SBI improves hardware utilization by splitting batches, it reduces the effective batch size seen by the GPU, limiting its benefit when batch sizes are small. As a result, SBI is effective only when very large batch sizes ($\geq$ 256) are sustained, as observed during the 26–35 s interval.

Fig. 10(b) shows the energy breakdown and J per token for the three systems. GPU+PIM with SBI achieves performance comparable to the GPU-only system but consumes more energy due to additional PIM. In contrast, GPU+PIM completes requests faster and achieves the lowest energy consumption, reducing J per token by 14.8%. The energy breakdown shows a modest increase in memory energy with PIM, which remains small compared to GPUs and CPUs. Overall, these results demonstrate that PIM can efficiently accelerate the decode phase of LLM serving with minimal energy overhead.

VIII. Related Work

Hardware-level and infrastructure-centric simulators. Hardware-centric simulation frameworks primarily model accelerator execution, device-level timing, and distributed communication. LLMCompass [54] evaluates accelerator and microarchitectural trade-offs for LLM workloads, and ADOR [55] explores hardware design choices for improving throughput. ASTRA-sim [56], [60] models communication hierarchy and collective synchronization at scale, while vTrain [61] analyzes parallelism strategies for training. These systems provide valuable visibility into microarchitectural execution and communication costs, but do not model request-driven inference serving dynamics, such as batch formation, prefix reuse, KV cache behavior, or end-to-end latency under workload variation. LLMServingSim 2.0 complements this line of work by extending analysis beyond accelerator behavior to serving-time interactions and performance evolution.

System-level LLM serving simulators. System-level simulators evaluate inference scheduling, batching, and deployment configuration. Vidur [26] and APEX [27] focus on latency/throughput prediction using operator profiling and configuration-space exploration for multi-device clusters. TokenSim [28] models dynamic arrival patterns and attention-state reuse, but relies on simplified memory abstractions that overlook bandwidth contention and KV cache movement overhead, limiting fidelity under realistic inference-serving dynamics. LLMServingSim [25] provides runtime statistics, including batch evolution and operator breakdowns, but supports only a single instance and does not support distributed KV cache placement or MoE-aware routing. In contrast, LLMServingSim 2.0 supports heterogeneous device pools, profile-based execution using real or simulated hardware, and detailed modeling of KV cache movement, prefix reuse, and expert-parallel routing, allowing full-stack evaluation of serving behavior under realistic, dynamic workloads.

IX. Conclusion

This paper addresses a fundamental challenge in modern LLM serving: understanding and reasoning about the complex interactions between increasingly heterogeneous hardware platforms and disaggregated serving architectures. As LLM serving systems evolve beyond homogeneous, scale-out deployments, effective system design requires tools that can capture how hardware characteristics, serving techniques, and runtime dynamics jointly shape performance, efficiency, and scalability. LLMServingSim 2.0 responds to this need by providing a unified simulation framework that makes these interactions explicit, analyzable, and extensible, enabling researchers and system designers to explore architectural trade-offs without the cost and rigidity of real deployments. By bridging hardware innovation and serving-system design within a single modeling framework, LLMServingSim 2.0 aims to serve as a practical foundation for both academic research and industrial exploration of next-generation LLM serving infrastructures.

Acknowledgments

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) (No.RS-2024-00396013), Electronics and Telecommunications Research Institute (ETRI) (No.RS-2025-02305453), and funded by the Korea government (MSIT). This work was also partly supported by SK hynix.

REFERENCES

[1] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson, "TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings," 2023. [Online]. Available: https://arxiv.org/abs/2304.01433

[2] Amazon Web Services, "Amazon Inferentia," https://aws.amazon.com/machine-learning/inferentia/, 2024, accessed: 2025-12-15.

[3] R. Prabhakar, R. Sivaramakrishnan, D. Gandhi, Y. Du, M. Wang, X. Song, K. Zhang, T. Gao, A. Wang, X. Li *et al.*, "Sambanova sn40l: Scaling the ai memory wall with dataflow and composition of experts," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 1353–1366.

[4] S. Lie, "Cerebras Architecture Deep Dive: First Look Inside the Hardware/Software Co-Design for Deep Learning," *IEEE Micro*, vol. 43, no. 3, pp. 18–30, 2023.

[5] D. Abts, J. Kim, G. Kimmell, M. Boyd, K. Kang, S. Parmar, A. Ling, A. Bitar, I. Ahmed, and J. Ross, " The Groq Software-defined Scale-out Tensor Streaming Multiprocessor : From chips-to-systems architectural overview ," in *2022 IEEE Hot Chips 34 Symposium (HCS)*. Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2022, pp. 1–69. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HCS55958.2022.9895630

[6] Rebellions Inc., "ATOM™ Architecture: Finding the Sweet Spot for GenAI," Rebellions Inc., White Paper, 2024. [Online]. Available: https://rebellions.ai/wp-content/uploads/2024/07/ATOMgenAI_white-paper.pdf

[7] H. Kim, Y. Choi, J. Park, B. Bae, H. Jeong, S. M. Lee, J. Yeon, M. Kim, C. Park, B. Gu, C. Lee, J. Bae, S. Bae, Y. Cha, W. Choe, J. Choi, J. Ha, H. Han, N. Hwang, S. Hwang, K. Jang, H. Je, H. Jeon, J. Jeon, H. Jeong, Y. Jung, D. Kang, H. Kim, M. Kim, M. Kim, S. Kim, S. Kim, W. Kim, Y. Kim, Y. Kim, Y. Ku, J. K. Lee, J. Lee, K. Lee, S. Lee, M. Noh, H. Oh, G. Park, S. Park, J. Seo, J. Seong, J. Paik, N. P. Lopes, and S. Yoo, "TCP: A Tensor Contraction Processor for AI Workloads Industrial Product*," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 890–902.

[8] S. Moon, J.-H. Kim, J. Kim, S. Hong, J. Cha, M. Kim, S. Lim, G. Choi, D. Seo, J. Kim, H. Lee, H. Park, R. Ko, S. Choi, J. Park, J. Lee, and J.-Y. Kim, "A Latency Processing Unit: A Latency-Optimized and Highly Scalable Processor for Large Language Model Inference," *IEEE Micro*, vol. 44, no. 6, pp. 17–33, 2024.

[9] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 43–56.

[10] S.-S. Park, K. Kim, J. So, J. Jung, J. Lee, K. Woo, N. Kim, Y. Lee, H. Kim, Y. Kwon, J. Kim, J. Lee, Y. Cho, Y. Tai, J. Cho, H. Song, J. H. Ahn, and N. S. Kim, "An LPDDR-based CXL-PNM Platform for TCO-efficient Inference of Transformer-based Large Language Models," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 970–982.

[11] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. Vijaykumar, "Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 372–385.

[12] S. Choi, M. Rhee, E. Kim, K. Shin, Y. Joo, and H. Kim, "PNM Meets Sparse Attention: Enabling Multi-Million Tokens Inference at Scale," *IEEE Computer Architecture Letters*, 2025.

[13] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. 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)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 193–210. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[14] P. Patel, E. Choukse, C. Zhang, A. Shah, Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient Generative LLM Inference Using Phase Splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 118–132.

[15] S. Rajbhandari, C. Li, Z. Yao, M. Zhang, R. Y. Aminabadi, A. A. Awan, J. Rasley, and Y. He, "DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale," 2022. [Online]. Available: https://arxiv.org/abs/2201.05596

[16] Z. Du, S. Li, Y. Wu, X. Jiang, J. Sun, Q. Zheng, Y. Wu, A. Li, H. H. Li, and Y. Chen, "SiDA: Sparsity-Inspired Data-Aware Serving for Efficient and Scalable Large Mixture-of-Experts Models," in *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., vol. 6, 2024, pp. 224–238. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2024/file/698cfaf72a208aef2e78bcac55b74328-Paper-Conference.pdf

[17] S. Cao, S. Liu, T. Griggs, P. Schafhalter, X. Liu, Y. Sheng, J. E. Gonzalez, M. Zaharia, and I. 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*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 715–730. [Online]. Available: https://doi.org/10.1145/3669940.3707267

[18] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo, "{Cost-Efficient} large language model serving for multi-turn conversations with {CachedAttention}," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 111–126.

[19] I. Gim, G. Chen, S.-s. Lee, N. Sarda, A. Khandelwal, and L. Zhong, "Prompt cache: Modular attention reuse for low-latency inference," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 325–338, 2024.

[20] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang, "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion," in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 94–109. [Online]. Available: https://doi.org/10.1145/3689031.3696098

[21] H. Kim, N. Wang, Q. Xia, J. Huang, A. Yazdanbakhsh, and N. S. Kim, "LIA: A Single-GPU LLM Inference Acceleration with Cooperative AMX-Enabled CPU-GPU Computation and CXL Offloading," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 544–558.

[22] X. Pan, E. Li, Q. Li, S. Liang, Y. Shan, K. Zhou, Y. Luo, X. Wang, and J. Zhang, "InstAttention: In-Storage Attention Offloading for Cost-Effective Long-Context LLM Inference," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 1510–1525.

[23] M. Seo, J. Hyun, S. Jeong, X. T. Nguyen, H.-J. Lee, and H. Lee, "OASIS: Outlier-Aware KV Cache Clustering for Scaling LLM Inference in CXL Memory Systems," *IEEE Computer Architecture Letters*, 2025.

[24] K. Kim, O. Kwon, Y. Park, and J. W. Lee, "AiDE: Attention-FFN Disaggregated Execution for Cost-Effective LLM Decoding on CXL-PNM," *IEEE Computer Architecture Letters*, vol. 24, no. 2, pp. 285–288, 2025.

[25] J. Cho, M. Kim, H. Choi, G. Heo, and J. Park, "LLMServingSim: A HW/SW Co-Simulation Infrastructure for LLM Inference Serving at Scale," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, 2024, pp. 15–29.

[26] A. Agrawal, N. Kedia, J. Mohan, A. Panwar, N. Kwatra, B. S. Gulavani, R. Ramjee, and A. Tumanov, "Vidur: A Large-Scale Simulation Framework For LLM Inference," in *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., vol. 6, 2024, pp. 351–366. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2024/file/b74a8de47d2b3c928360e0a011f48351-Paper-Conference.pdf

[27] Y.-C. Lin, W. Kwon, R. Pineda, and F. N. Paravecino, "APEX: An Extensible and Dynamism-Aware Simulator for Automated Parallel Execution in LLM Serving," 2025. [Online]. Available: https://arxiv.org/abs/2411.17651

[28] F. Wu, Z. Bian, G. Duan, T. Xu, J. Wu, T. Ma, Y. Yao, R. Gong, and Y. Zhuo, "TokenSim: Enabling Hardware and Software Exploration for Large Language Model Inference Systems," in *Advanced Parallel Processing Technologies: 16th International Symposium, APPT 2025, Athens, Greece, July 13-16, 2025, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2025, p. 257–266. [Online]. Available: https://doi.org/10.1007/978-981-95-1021-4_19

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[30] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. 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), 2023, pp. 663–679.

[31] J. Hu, J. Xu, Z. Liu, Y. He, Y. Chen, H. Xu, J. Liu, J. Meng, B. Zhang, S. Wan, G. Dan, Z. Dong, Z. Ren, C. Liu, T. Xie, D. Lin, Q. Zhang, Y. Yu, H. Feng, X. Chen, and Y. Shan, "DEEPSERVE: serverless large language model serving at scale," in *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '25. USA: USENIX Association, 2025.

[32] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "Spotserve: Serving generative large language models on preemptible instances," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1112–1127.

[33] Y. Xiang, X. Li, K. Qian, Y. Yang, D. Zhu, W. Yu, E. Zhai, X. Liu, X. Jin, and J. Zhou, "Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, ser. SOSP '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1030–1045. [Online]. Available: https://doi.org/10.1145/3731569.3764815

[34] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2021, pp. 1–15.

[35] R. Hwang, J. Wei, S. Cao, C. Hwang, X. Tang, T. Cao, and M. Yang, "Pre-gated MoE: An Algorithm-System Co-Design for Fast and Scalable Mixture-of-Expert Inference," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1018–1031.

[36] W. Chen, S. He, H. Qu, R. Zhang, S. Yang, P. Chen, Y. Zheng, B. Huai, and G. Chen, "IMPRESS: An Importance-Informed Multi-Tier Prefix KV Storage System for Large Language Model Inference," in *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. Santa Clara, CA: USENIX Association, Feb. 2025, pp. 187–201. [Online]. Available: https://www.usenix.org/conference/fast25/presentation/chen-weijian-impress

[37] J. Wang, J. Han, X. Wei, S. Shen, D. Zhang, C. Fang, R. Chen, W. Yu, and H. Chen, "KVCache cache in the wild: characterizing and optimizing KVCache cache at a large cloud provider," in *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '25. USA: USENIX Association, 2025.

[38] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "SGLang: efficient execution of structured language model programs," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, ser. NIPS '24. Red Hook, NY, USA: Curran Associates Inc., 2024.

[39] Y. Liu, Y. Cheng, J. Yao, Y. An, X. Chen, S. Feng, Y. Huang, S. Shen, R. Zhang, K. Du, and J. Jiang, "LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference," 2025. [Online]. Available: https://arxiv.org/abs/2510.09665

[40] R. Qin, Z. Li, W. He, J. Cui, F. Ren, M. Zhang, Y. Wu, W. Zheng, and X. 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)*. Santa Clara, CA: USENIX Association, Feb. 2025, pp. 155–170. [Online]. Available: https://www.usenix.org/conference/fast25/presentation/qin

[41] Y. Jin, T. Wang, H. Lin, M. Song, P. Li, Y. Ma, Y. Shan, Z. Yuan, C. Li, Y. Sun, T. Wu, X. Chu, R. Huan, L. Ma, X. You, W. Zhou, Y. Ye, W. Liu, X. Xu, Y. Zhang, T. Dong, J. Zhu, Z. Wang, X. Ju, J. Song, H. Cheng, X. Li, J. Ding, H. Guo, and Z. Zhang, "P/D-Serve: Serving Disaggregated Large Language Model at Scale," 2024. [Online]. Available: https://arxiv.org/abs/2408.08147

[42] S. Hong, S. Moon, J. Kim, S. Lee, M. Kim, D. Lee, and J.-Y. Kim, "DFX: A Low-latency Multi-FPGA Appliance for Accelerating Transformer-based Text Generation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 616–630.

[43] H. Wang, Z. Wang, Z. Yue, Y. Long, T. Wei, J. Yang, Y. Wang, C. Li, S. Wei, Y. Hu, and S. Yin, "MCBP: A Memory-Compute Efficient LLM Inference Accelerator Leveraging Bit-Slice-enabled Sparsity and Repetitiveness," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1592–1608. [Online]. Available: https://doi.org/10.1145/3725843.3756037

[44] S. Moon, J. Cha, H. Park, and J.-Y. Kim, "Hybe: GPU-NPU Hybrid System for Efficient LLM Inference with Million-Token Context Window," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 808–820. [Online]. Available: https://doi.org/10.1145/3695053.3731051

[45] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham, G. Kim, D. Mahajan, and J. Park, "NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 722–737. [Online]. Available: https://doi.org/10.1145/3620666.3651380

[46] J. Park, J. Choi, K. Kyung, M. J. Kim, Y. Kwon, N. S. Kim, and J. H. Ahn, "AttAcc! Unleashing the Power of PIM for Batched Transformer-based Generative Model Inference," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 103–119. [Online]. Available: https://doi.org/10.1145/3620665.3640422

[47] S. Yun, K. Kyung, J. Cho, J. Choi, J. Kim, B. Kim, S. Lee, K. Sohn, and J. H. Ahn, " Duplex: A Device for Large Language Models with Mixture of Experts, Grouped Query Attention, and Continuous Batching ," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2024, pp. 1429–1443. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MICRO61859.2024.00105

[48] Y. He, H. Mao, C. Giannoula, M. Sadrosadati, J. Gómez-Luna, H. Li, X. Li, Y. Wang, and O. Mutlu, "Papi: Exploiting dynamic parallelism in large language model decoding with a processing-in-memory-enabled computing system," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 766–782.

[49] D. Gouk, S. Kang, S. Lee, J. Kim, K. Nam, E. Ryu, S. Lee, D. Kim, J. Jang, H. Bae, and M. Jung, "CXL-GPU: Pushing GPU Memory Boundaries with the Integration of CXL Technologies," *IEEE Micro*, pp. 1–8, 2025.

[50] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access,{High-Performance} memory disaggregation with {DirectCXL}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 287–294.

[51] J. Liu, H. Hadian, Y. Wang, D. S. Berger, M. Nguyen, X. Jian, S. H. Noh, and H. Li, "Systematic cxl memory characterization and performance analysis at scale," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 1203–1217.

[52] W. Lee, J. Lee, J. Seo, and J. Sim, "{InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 155–172.

[53] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 094–31 116.

[54] H. Zhang, A. Ning, R. B. Prabhakar, and D. Wentzlaff, "LLMCompass: Enabling Efficient Hardware Design for Large Language Model Inference," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1080–1096.

[55] J. Kim, H. Lee, G. Ko, G. Choi, S. Ham, S. Hong, and J.-Y. Kim, "ADOR: A Design Exploration Framework for LLM Serving with Enhanced Latency and Throughput," in *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2025, pp. 15–25.

[56] W. Won, T. Heo, S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "ASTRA-sim2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-model Training at Scale," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 283–294.

[57] S. Sridharan, T. Heo, L. Feng, Z. Wang, M. Bergeron, W. Fu, S. Zheng, B. Coutinho, S. Rashidi, C. Man, and T. Krishna, "Chakra: Advancing Performance Benchmarking and Co-design using Standardized Execution Traces," 2023. [Online]. Available: https://arxiv.org/abs/2305.14516

[58] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 611–626. [Online]. Available: https://doi.org/10.1145/3600006.3613165
[59] ShareGPT Team, "ShareGPT," https://sharegpt.com, 2023.
[60] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 81–92.
[61] J. Bang, Y. Choi, M. Kim, Y. Kim, and M. Rhu, "vTrain: A Simulation Framework for Evaluating Cost-Effective and Compute-Optimal Large Language Model Training," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 153–167.

APPENDIX

A. Abstract

LLMServingSim 2.0 is a unified, runtime-driven simulator for heterogeneous and disaggregated LLM inference serving infrastructures, implemented in Python and C++. Given system configurations and request traces, it models scheduling, data movement, and contention in server-scale systems to estimate throughput and latency breakdowns. The simulator additionally supports an integrated power/energy model, MoE inference, prefill-decode disaggregation and memory disaggregation, multi-tier prefix caching, and emerging hardware platforms such as TPU, PIM, and CXL. This artifact packages the simulator and scripts/configurations needed to reproduce the key experimental results presented in the paper.

B. Artifact check-list (meta-information)

- **Compilation:** gcc/g++ 11.4.0
- **Model:** Llama 3.1-8B/70B, Phi-mini MoE, Mixtral 8×7B
- **Data set:** ShareGPT [59]
- **Run-time environment:** Ubuntu 22.04 LTS, Linux kernel 6.8.0-100-generic
- **Hardware:** x86-64
- **Execution:**
```
$ cd evaluation
$ ./run_all.sh
$ ./compare.sh
```
- **Metrics:** Throughput (tokens/s), latency (TTFT, TPOT, ITL, incl. p99), prefix hit rate (%), power (W), energy (J), memory usage (MB)
- **Output:** standard output, CSV files
- **Experiments:** Reproduce evaluation results for Figures 5-10
- **How much disk space required (approximately)?:** 15GB
- **How much time is needed to prepare workflow (approximately)?:** 5 minutes
- **How much time is needed to complete experiments (approximately)?:** 2 hours 30 minutes
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Creative Commons Attribution 4.0 International, MIT License
- **Workflow automation framework used?:** No
- **Archived (provide DOI)?:** Yes (10.5281/zenodo.18879965)

C. Description

1) How to access

- Zenodo: LLMServingSim 2.0 is published on Zenodo: https://doi.org/10.5281/zenodo.18879965
- GitHub: LLMServingSim 2.0 is available on GitHub: https://github.com/casys-kaist/LLMServingSim

2) Hardware dependencies

LLMServingSim 2.0 requires an x86-64 architecture, and the simulation time may be affected by hardware differences. For similar simulation time results, we recommend using the hardware specified in Section VI.

3) Software dependencies

LLMServingSim 2.0 has been tested on Ubuntu 22.04 LTS with Python 3.10.12 and requires gcc and g++ versions 11.4.0 or higher. Additionally, it requires the software prerequisites of ASTRA-Sim [56] and Chakra [57]. To meet these software prerequisites, we provide a prebuilt Docker image. Reviewers can download and run the image using the provided scripts. See Appendix D for details.

4) Data sets

We use the ShareGPT [59] dataset to generate request traces synthesized with a Poisson arrival process.

5) Models

We use Llama 3.1-8B/70B, Phi-mini MoE, and Mixtral 8×7B for our evaluation. Their model architectures follow decoder-only transformer variants, including both dense and MoE designs.

D. Installation

- Clone the LLMServingSim 2.0 repository.
```
$ git clone --recurse-submodules https://
    github.com/casys-kaist/LLMServingSim.
    git
$ cd LLMServingSim
```
- Run Docker.
```
$ ./docker.sh
```
- Build submodules.
```
$ ./compile.sh
```

E. Experiment workflow

The workflow of LLMServingSim 2.0 is described in Section IV, particularly in Fig. 1. The simulator takes (1) a workload configuration, (2) a cluster configuration, and (3) hardware performance profiles. The hardware performance profiles are used by each MSG when generating execution graphs. During a runtime-driven loop, the serving engine routes requests to the appropriate MSG, performs dynamic batching, and generates an execution graph under the configured serving policies, including weight offloading, expert routing, prefill-decode mapping, KV cache transfer, and prefix caching. The System Simulator then evaluates the graph while modeling communication, contention, multi-tier memory accesses, and integrated power/energy, and returns the results to advance the next iteration. At each iteration, LLMServingSim 2.0 reports throughput, prefix hit rate, power/energy, and memory usage. Finally, it aggregates the overall results and reports per-request latency metrics.

F. Evaluation and expected results

As described in Section VII, our artifact evaluation consists of six experiments corresponding to Fig. 5 through Fig. 10. To reproduce them, we provide one script per figure (`figure_{i}.sh`) in the `evaluation/` folder. We also provide `run_all.sh` to execute the full evaluation pipeline at once.

- Move to `evaluation/` folder.
```
$ cd evaluation
```
- Run each evaluation one by one.
```
$ ./figure_5.sh
$ ./figure_6.sh
...
$ ./figure_10.sh
```
- Run all evaluation at once.
```
$ ./run_all.sh
```

Each script stores outputs in its corresponding folder (`figure_5/` to `figure_10/`). During execution, it generates `logs/`, `results/`, and `parsed/` subdirectories, and produces the final figure PDFs (`figure_{i}.pdf`). The

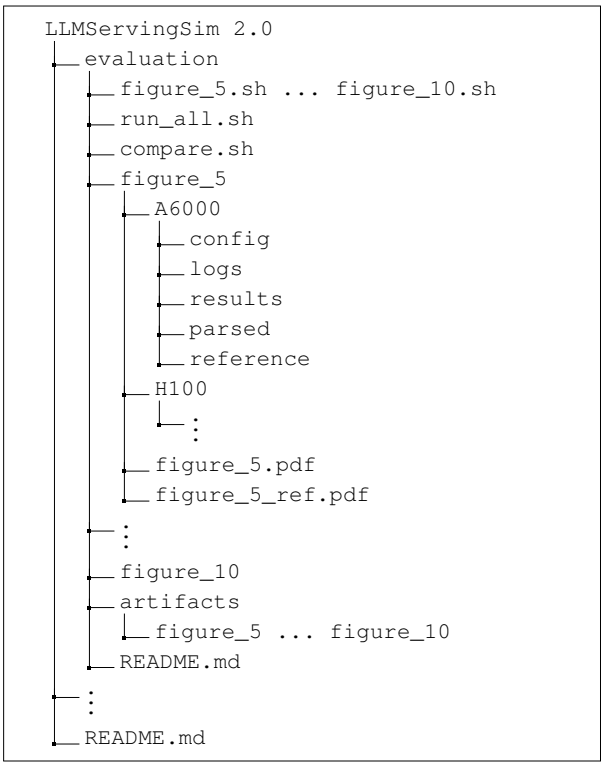


Fig. 11. Directory tree of LLMServingSim 2.0 evaluation.

parsed TSV files contain metrics such as throughput, latency, simulation time, memory, or power, depending on the figure.

For verification, we provide preserved reference outputs in `evaluation/artifacts/` and reference figures (`figure_{i}_ref.pdf`) in each figure folder. To compare numerical values of the evaluation, we provide `compare.sh`, which automatically compares generated parsed TSV files against the reference outputs in `evaluation/artifacts/figure_{i}/parsed`.

- Compare one or more figures.
  ```
  $ ./compare.sh 5
  $ ./compare.sh 6 7 8
  $ ./compare.sh figure_9
  ```
- Compare all figures at once.
  ```
  $ ./compare.sh
  ```

For visual comparison, compare each generated figure PDF (`figure_{i}_.pdf`) with its corresponding reference PDF (`figure_{i}_ref.pdf`) in the same figure folder. For more information about evaluation, please refer to `evaluation/README.md`. Additional per-figure details are documented in `evaluation/figure_{i}/README.md`, including each figure's objective, required configurations and datasets, exact run commands, expected generated PDF/TSV outputs, and numerical/visual validation procedures.

Fig. 11 illustrates the directory tree of LLMServingSim 2.0, including the per-figure evaluation scripts (`figure_{i}.sh`), their corresponding `figure_{i}` folders with generated outputs (`logs/`, `results/`, `parsed/`, and PDFs), the utility scripts (`run_all.sh`, `compare.sh`), the archived reference outputs in `artifacts/`, reference figure PDFs (`figure_{i}_ref.pdf`), and `README.md` files.

G. Experiment customization

1) Input configurations

LLMServingSim 2.0 uses a cluster configuration JSON file as the main hardware/system input. Cluster configuration is located in `cluster_config/{name}.json`, where users can customize topology and instance settings such as `num_nodes`, `link_bw`, `link_latency`, `num_instances`, `cpu_mem`, `model_name`, `hardware`, `npu_mem`, `npu_num`, `npu_group`, and `pd_type`. Optional fields include `placement`, `pim_config`, `power`, and `cxl_mem`.

2) Input dataset

LLMServingSim 2.0 uses request traces in JSONL format located in `dataset/{name}.jsonl`. Each request line includes `input_toks`, `output_toks`, `arrival_time_ns`, and `input_tok_ids`. Custom traces can be generated with `dataset/sharegpt_parser.py` or created manually using the same JSONL schema.

3) Input parameters

LLMServingSim 2.0 `main.py` provides runtime options listed below. See `README.md` for more details.

- Input/output options:
  `--cluster-config`, `--dataset`, `--output`
- Core options:
  `--fp`, `--block-size`, `--max-batch`,
  `--max-num-batched-tokens`, `--num-req`
- Routing/scheduling options:
  `--request-routing-policy`
  `--expert-routing-policy`
  `--prioritize-prefill`
- Feature toggles:
  `--enable-prefix-caching`
  `--enable-prefix-sharing`
  `--prefix-storage`
  `--enable-local-offloading`
  `--enable-attn-offloading`
  `--enable-sub-batch-interleaving`
  `--enable-attn-prediction`
- Run-control/logging options:
  `--gen`, `--log-interval`, `--log-level`
  `--network-backend`

4) Evaluation-script customization

For evaluation, each per-figure script (`figure_{i}.sh`) declares key inputs near the top of the file, including paths to cluster configurations and datasets, as well as runtime options passed to `main.py`. Users can customize configuration files, workload traces, and execution settings using the input parameters listed in Appendix G3.

H. Notes

More information can be found in the `README.md` file of each directory.