rLLM: A Serving-Aware Simulation Framework for Realistic LLM Hardware Evaluation

Shiyi Liu*
shiyi.liu@connect.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong

Yiran Xia*
yxiabl@connect.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong

Zhenhua Zhu
zhuzhenhua@mail.tsinghua.edu.cn
Tsinghua University
Beijing, China
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong

Chen Bai
cbai@fudan.edu.cn
Institute of Systems for Advanced Computing at Fudan University (FiSAC)
Fudan University
Shanghai, China

Ceyu Xu†
eeentropy@ust.hk
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong

Yuan Xie
yuanxie@ust.hk
The Hong Kong University of Science and Technology
Hong Kong, Hong Kong

Abstract

Large language model inference has become a major target for computer architecture research, driving a wide range of work on specialized accelerators, serving systems, and emerging technologies. However, prior studies usually evaluate inference hardware under simplified assumptions, such as static request shapes or fixed execution patterns. These assumptions divorce hardware evaluation from actual serving dynamics, ultimately leading to inaccurate performance estimates and suboptimal architectural conclusions.

We present rLLM, a zero-semantic-gap framework for realistic LLM hardware evaluation. rLLM is implemented as an extension to the official vLLM codebase, allowing it to seamlessly track upstream vLLM updates while interoperating with other vLLM extensions. By coupling the real vLLM software stack with our pluggable hardware simulators, rLLM enables end-to-end evaluation of LLM accelerators driven by realistic, highly dynamic serving behaviors without requiring full platform-specific vLLM re-implementations.

We validate rLLM's end-to-end fidelity using profiling-based, analytical, and microarchitecture backend configurations. Designed with modularity in mind, the framework is easily extensible to support a wide range of emerging accelerators. Crucially, our exploration reveals that realistic serving dynamics fundamentally shift system bottlenecks and optimal design points compared to static evaluations. By uncovering these discrepancies, rLLM offers the community fresh design insights from a serving perspective, underscoring the necessity of serving-aware co-simulation for robust LLM hardware co-design.

CCS Concepts: • **Computing methodologies** → **Discrete-event simulation**; *Machine learning*; Parallel computing methodologies.

Keywords: large language model inference serving, hardware-software co-simulation, vLLM, design space exploration, performance modeling

ACM Reference Format:
Shiyi Liu, Yiran Xia, Zhenhua Zhu, Chen Bai, Ceyu Xu, and Yuan Xie. 2027. rLLM: A Serving-Aware Simulation Framework for Realistic LLM Hardware Evaluation. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3845814.3850087

1 Introduction

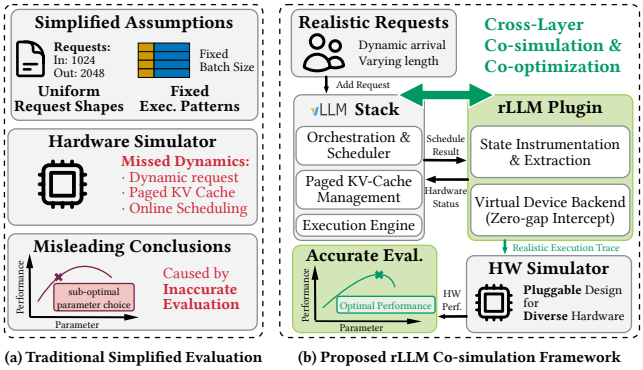
With the explosive growth of generative artificial intelligence, large language model (LLM) inference has rapidly emerged as one of the most dominant computational tasks in global data centers, significantly outpacing traditional IT workloads and other AI operations in both scale and operational cost [8, 9, 12, 25, 29, 31]. To make LLM inference cheaper, faster, more scalable, and more sustainable, computer architecture and systems researchers have addressed

*Both authors contributed equally to this research.
†Corresponding author.



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

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850087



(a) Traditional Simplified Evaluation (b) Proposed rLLM Co-simulation Framework

Figure 1. Overview of rLLM. Implemented as a plugin on top of the vLLM stack, rLLM extracts serving state and workload behavior from dynamic user requests, translates them into aggregated traces, and drives downstream architectural and hardware simulators for realistic end-to-end evaluation.

this problem from numerous angles, ranging from fabrication [32], emerging devices [21], packaging technologies [22] to architectural [17], compiler [7], algorithmic [37], and orchestration [20].

However, the complex and tightly coupled nature of the LLM serving stack makes accurate performance evaluation inherently challenging. For researchers whose expertise lies primarily in a single layer of the stack, evaluating the full system requires substantial effort across hardware, architecture, runtime, and algorithmic considerations. As a result, new ideas are often evaluated under simplified assumptions that capture only part of the real deployment. This issue is particularly limiting in architecture research, where detailed architectural evaluation is already time-consuming and existing tools rarely connect lower-layer hardware and architectural behaviors with upper-layer orchestration mechanisms.

At the same time, this challenge makes holistic evaluation more important than ever for computer architecture research, a field fundamentally concerned with the interaction between software and hardware. High-fidelity evaluation of the end-to-end LLM inference stack is increasingly necessary because upper-layer software decisions, including orchestration frameworks such as vLLM and parallel execution strategies, can substantially reshape how inference workloads are mapped onto lower-level architectural resources and hardware execution behavior.

In this work, we aim to bridge this gap by introducing rLLM. rLLM stands for realistic LLM, and is designed to faithfully carry details from model-parallel execution strategies, orchestration behaviors, and dynamic request patterns into downstream architectural and hardware simulation, as illustrated in Figure 1. Rather than recreating an entirely new orchestration layer, rLLM is implemented as a plugin on top of vLLM. The plugin integration allows rLLM to work naturally with the broader software ecosystem around vLLM, track the evolution of vLLM over time, and preserve high simulation fidelity with a minimal semantic gap.

As illustrated in Figure 1 (b), rLLM enables end-to-end co-simulation by tightly coupling the vLLM inference engine with lower-tier architectural and hardware simulators. Concretely, rLLM works in three stages. 1) It replays a pre-collected user request trace through vLLM and provides a virtual backend, allowing vLLM to execute its native request decomposition, scheduling, and runtime logic without requiring physical hardware execution. 2) It intercepts the operator-level workloads generated by vLLM and exposes them to lower-tier architectural and hardware simulators, which return performance feedback that rLLM back-pressures vLLM to achieve the correct simulated time tick. 3) To faithfully model paged attention, rLLM provides a virtual memory pool for vLLM's paged KV cache management, enabling faithful memory traces to be collected and forwarded to lower-tier simulators for accurate performance and energy analysis. Together, these mechanisms allow rLLM to preserve realistic inference-engine behavior while supporting detailed cross-layer evaluation.

rLLM also provides computer architects with a clean integration interface. To enable simulation for a new hardware accelerator, a hardware designer only needs to implement a small set of kernels, such as attention and GEMM, together with lightweight callback interfaces for modeling communication and memory access. This interface substantially lowers the effort required to connect detailed accelerator models with a realistic serving stack. Without rLLM, architects are often limited to answering narrow questions under highly simplified assumptions, such as "What is the performance and energy of serving a request of a particular shape at a fixed batch size?" By coupling rLLM with an accelerator's architectural and hardware simulation backend, the evaluation scope becomes much broader and more realistic. Architects can now ask questions such as "What is the time to first token (TTFT) under a realistic workload of 1,000 user requests per second?", "What is the maximum throughput, and what parallelization strategy should be used, if the system must keep TTFT within one second?", and "For a given serving stack and hardware platform, how do agentic coding workloads differ from long-prefill document-understanding workloads in performance behavior?"

rLLM is the first simulation bridge that exposes the dynamics of real workloads and orchestration policies to architectures, enabling joint exploration of hardware design, serving parameters, and workload characteristics within a unified evaluation framework. In fact, through our exploration and case studies of applying rLLM to existing accelerator designs, we find that accounting for workload dynamism is more important than the community has generally assumed: optimizing for simplified static workload assumptions is often misleading, **whereas design-space exploration that**

explicitly targets realistic dynamism can lead to substantially different configurations and conclusions.

We make the following contributions in this work.

- We identify a critical gap in existing LLM accelerator evaluation. Prior approaches do not faithfully connect dynamic serving workloads and orchestration behavior with lower-level architectural and hardware simulation.
- We build rLLM, a plugin-based co-simulation framework on top of vLLM. rLLM preserves realistic serving semantics while exposing operator-, communication-, and memory-level behaviors to downstream simulators. To the best of our knowledge, rLLM is the first framework that enables unified cross-layer evaluation of LLM serving dynamics together with detailed accelerator simulation. We validate its end-to-end fidelity using profiling-based, analytical, and microarchitecture backend configurations.
- Through case studies on existing accelerator designs, we show that incorporating realistic workload dynamism materially changes evaluation outcomes, design choices, and optimization conclusions compared with conventional static workload assumptions.
- We will open-source rLLM upon the acceptance of this paper to facilitate further research in serving-aware LLM hardware co-design.

2 Background

2.1 Status of LLM Accelerator Evaluation

Computer architects have proposed numerous specialized accelerators for LLM workloads. Leveraging advanced technologies such as CXL-based memory expansion and near-memory platforms [13, 19, 28, 30], Processing-In-Memory [4, 10, 16, 27], and 3D-integrated or hybrid-bonded memory-compute stacks [11, 14, 22, 26], these designs report substantial performance speedups over conventional GPU inference systems. However, the evaluation methodologies employed by many of these accelerators are often overly simplified, failing to capture the complexity of real-world LLM serving scenarios.

Specifically, many evaluations rely on static batch sizes and sequence lengths, simplifying the actual serving scenario into static kernel graphs and fixed operator bindings [10, 22, 23, 27]. Other recent works have recognized the necessity of addressing serving dynamism. NeuPIMs [16] proposes sub-batch interleaving to accommodate fluctuating operational intensities, PAPI [15] proposes a custom online scheduler to dynamically orchestrate GPU-PIM coordination. While these approaches represent a step forward, their modeling of serving dynamics captures only a subset of crucial system behaviors. Furthermore, requiring architects to independently re-implement highly complex software stacks into isolated hardware simulators is not only error-prone but fundamentally unscalable given the rapid evolution of state-of-the-art serving frameworks.

2.2 vLLM and SGLang Inference Engine

Modern LLM serving engines do far more than invoke model kernels. LLM inference requests have highly dynamic shapes because prompt lengths and decode lengths vary across users and across time. This variability is especially problematic during auto-regressive decoding, whose arithmetic intensity is low and whose performance is often limited by memory movement rather than raw compute throughput. As a result, processing requests independently often fails to saturate both on-chip compute resources and memory bandwidth, leading to poor hardware utilization and low serving efficiency.

Frameworks such as vLLM and SGLang address this problem through request orchestration, continuous batching, prefix-caching and KV cache management [20, 40]. By co-batching requests with heterogeneous prefill and decode behavior, they improve both compute utilization and memory-bandwidth utilization through increased batch size. These runtime decisions directly shape operator mix, request interleaving, memory footprint, communication behavior, and kernel timing, making them critical to both end-to-end serving performance and the hardware behavior seen by architectural simulators. In this work, we target vLLM because SGLang is also built upon key components of the vLLM execution stack, so a vLLM plugin naturally eases later support for SGLang.

2.3 Simulators for LLM Inference

Simulating LLM inference requires reasoning across multiple layers of the serving stack, from model execution and runtime scheduling to communication and memory behavior. Existing simulators each capture different parts of this design space. At the system and communication level, Astra-Sim and its successor Astra-sim2.0 provide infrastructure for modeling distributed execution, collective communication, and large-scale system behavior [35]. At the processor level, GPGPU-Sim offers detailed GPU micro-architectural simulation for kernel execution and memory interactions [3]. At the memory level, DRAMSim3 provides cycle-accurate modeling of DRAM timing, power, and thermal behavior [24]. These tools are valuable building blocks for architectural analysis, but they are not designed to directly model the end-to-end dynamics of modern LLM serving.

Recent LLM-specific simulators have started to close this gap. LLMCompass enables hardware design exploration for LLM inference through workload characterization, mapping, and accelerator-oriented architectural simulation [39]. However, its main goal is hardware design-space exploration rather than preserving the runtime semantics of an actual serving framework. LLMServingSim moves further toward end-to-end serving simulation by jointly modeling serving software and hardware behavior for LLM inference workloads [6]. Together, these prior efforts show the importance of cross-layer simulation for LLM inference, but they also highlight a remaining gap between production-grade serving

runtimes and downstream architectural simulators. This gap motivates rLLM, which is designed to preserve realistic vLLM runtime behavior while translating serving events and execution states into forms that can drive detailed lower-level simulation components.

3 Motivation

LLM inference systems have evolved into highly complex software stacks, often requiring millions of lines of code to realize their full runtime behavior. So do the hardware systems that run LLM inference, whose performance increasingly depends on intricate interactions across compute, memory, and communication. Today, orchestration layers such as vLLM have become a necessity for both edge and datacenter deployments, making accurate modeling of the interplay between the orchestration layer and model execution more important than ever. In this section, we introduce the key ideas behind rLLM and the motivations for its design through three guiding questions:

Q1. Why is evaluation without modeling the orchestration layer problematic? In modern LLM inference systems, the primary bottleneck is often memory bandwidth and inter-host communication bandwidth rather than raw compute. This makes request batching across users essential, as it improves weight reuse, amortizes kernel launches, and increases overall system utilization. In addition, production inference stacks rarely operate on a single GPU and instead rely on multiple forms of parallelism, including tensor parallelism, pipeline parallelism, data parallelism, and expert parallelism. These mechanisms materially affect execution behavior, system-level tradeoffs, and memory and communication traffic. Modern serving systems also employ mechanisms such as continuous batching, chunked prefilling, and KV cache offloading and scheduling, all of which strongly influence latency, throughput, and resource utilization. More importantly, the orchestration layer affects not only average throughput, but also tail latency, fairness, and service-level behavior across concurrent users. In practice, cache management is part of the execution semantics rather than a secondary optimization, since prefix reuse, block allocation, eviction, and offloading directly shape memory behavior and scheduling decisions. As a result, evaluating an architecture without modeling the orchestration layer can be misleading.

Q2. What causes the misalignment between static and dynamic evaluation? The misalignment arises because static evaluation assumes a fixed execution pattern, whereas real LLM serving continuously changes with workload dynamics. In practice, request shapes vary significantly across users and evolve over time, which changes batch composition, the prefill-decode balance, communication patterns, and memory pressure. Static evaluation also fails to capture the throughput-latency tradeoff induced by runtime scheduling, which changes with workload intensity and batch composition. Even requests with similar nominal shapes can exhibit different runtime behavior because prefix locality, cache hits, and cache misses change the effective amount of computation, memory traffic, and data movement. As a result, the system operating point is determined not only by request shapes, but also by how the serving framework groups, prioritizes, and stages requests over time. Therefore, an architecture that performs well for a single fixed request shape may not excel in realistic deployments, where the true operating conditions are neither static nor uniform.

Q3. Why implement rLLM as a vLLM plugin? Implementing rLLM as a native vLLM plugin is the most effective way to minimize the semantic gap between the simulated serving stack and real deployments. Modern serving frameworks evolve rapidly and contain complex runtime logic spanning scheduling, cache management, memory allocation, and parallel execution. Re-implementing these behaviors in a separate simulator frontend is not only labor-intensive, but also difficult to keep aligned with upstream software, which can easily introduce modeling errors. By building on top of the native vLLM plugin interface, rLLM preserves the real control flow of the production serving engine while overriding only the hardware-facing execution path. The plugin architecture is non-intrusive and extensible: it allows rLLM to track upstream vLLM updates, interoperate with the broader vLLM ecosystem, and expose a hardware-agnostic interface for integrating heterogeneous accelerator backends. As a result, the plugin architecture is not merely an implementation convenience, but a key design choice that makes realistic and maintainable hardware-software co-simulation possible.

Summary Taken together, these observations suggest that realistic LLM hardware evaluation requires not only accurate low-level simulation, but also faithful execution of the serving control plane. Ignoring this interaction does not merely introduce quantitative error in latency or throughput estimates, but can also change the design conclusions drawn from evaluation. We will show in Section 6 that serving-aware evaluation can lead to substantially different architectural rankings and better final design points than conventional static design space exploration (DSE).

4 rLLM Framework

To address the evaluation gap in existing architectural studies, we propose rLLM. This co-simulation framework enables realistic hardware evaluation for LLM serving. Figure 2 illustrates the high-level architecture and core execution flow of our design. The system faithfully captures system-level dynamics through a native vLLM scheduler frontend (c). At the core of rLLM lies the **rLLM frontend (d)** and the **rLLM multi-fidelity backend (e)**. The rLLM frontend, implemented as a vLLM platform plugin, gathers runtime scheduler decisions and intercepts key operators (Section 4.1). The intercepted

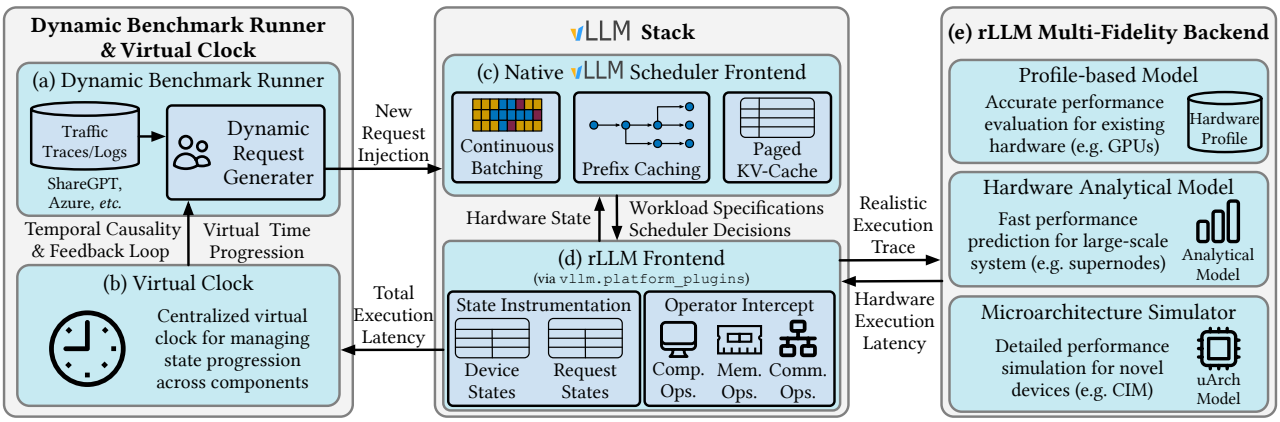


Figure 2. Overview of the rLLM architecture. (a) Dynamic benchmark runner generates requests according to realistic traffic patterns. (d) The rLLM frontend operates as a native plugin for (c) the native vLLM scheduler frontend. The rLLM frontend intercepts key operations on devices and sends the execution trace to the backend. (e) The Multi-Fidelity Backend evaluates these traces to estimate simulated target latency. (b) The virtual clock feeds the current virtual time back to the benchmark runner.

operators form a realistic execution trace and are not sent to hardware for execution. The multi-fidelity hardware backend processes these traces to estimate simulated target latency (Section 4.2). A virtual clock (b) then advances virtual time using the simulated target latency. Alongside this clock, a dynamic benchmark runner (a) orchestrates the simulation workflow. It injects inference requests according to realistic arrival patterns and virtual time (Section 4.3). We elaborate the execution flow and the lifecycle of a single request in Section 4.4.

4.1 rLLM as a vLLM Plugin

The rLLM frontend registers as a vLLM plugin (via `vllm.platform_plugins`) to achieve zero-intrusion interception of the inference execution flow, as illustrated in Figure 2 (d). Registering a virtualized execution backend allows rLLM to integrate natively with the unmodified vLLM engine (shown in Figure 2 (c)). To implement this mechanism, we inherit and override the base `Worker` and `ModelRunner` classes within the vLLM ecosystem. The custom `Worker` class acts as a transparent proxy that bypasses physical hardware execution while preserving the original communication and synchronization logic. The proxy design preserves high-level optimizations such as continuous batching and prefix caching during simulation.

The core of our state instrumentation logic resides in the overridden `ModelRunner` class. At initialization, rLLM retrieves device states such as total memory capacity directly from the simulator backend. The frontend utilizes these physical constraints to initialize a virtual memory pool. This pool allows the native vLLM scheduler to determine the exact number of available blocks for dynamic request batching. During runtime, the framework meticulously tracks the request states, including dynamic block allocations and page table updates managed by the PagedAttention mechanism. Capturing these precise virtual-to-physical memory mappings allows the hardware backend to accurately model spatial locality and memory fragmentation.

Beyond memory management, our framework natively intercepts all performance-critical computation operators and collective communication events. This interception occurs continuously throughout the forward inference pass. With the operator interception, rLLM extracts detailed execution semantics including exact memory access patterns, matrix dimensions, communication types, and data transfer volumes. We explicitly capture the complete set of input parameters that vLLM natively prepares for its GPU backend operators. These comprehensive parameters are then packaged into a fine-grained, hardware-agnostic trace. Downstream hardware simulators process this extracted trace to perform precise performance and energy evaluations. This approach exposes the exact runtime compute and communication demands to the backend without requiring physical accelerator execution.

By intercepting workloads at the application platform layer, rLLM provides a device-agnostic abstraction for architectural exploration. Traditional tools relying on OS-level interception often force a GPU-centric memory hierarchy upon the simulation [2]. In contrast, our plugin-based approach decouples the serving logic from the underlying hardware implementation. Architects can seamlessly evaluate heterogeneous designs such as processing-in-memory accelerators by overriding the Python-level memory allocation classes. This application-level flexibility enables researchers to explore custom hardware-specific key-value cache placement algorithms. Therefore, the rLLM frontend serves as a robust bridge that carries dynamic serving semantics into low-level design space exploration.

```python
class SimulatorA(SimBackend):
    simulateStep(ctx: StepContext) -> StepResult:
        runLinear(ctx.linearCtx)   -> latLinear
        runAttention(ctx.attnCtx)  -> latAttn
        runComm(ctx.commCtx)       -> latComm

        latLinear + latAttn + latComm -> totalLatency
        return StepResult(totalLatency)
```

Code 1. Pseudocode for integrating `SimulatorA`

It is worth noting that the rLLM frontend incurs CPU time for recurring control-plane operations such as request scheduling and KV cache management, which we account for as CPU/software latency. We include the CPU/software latency incurred during each serving step, but exclude one-time warm-up costs such as process startup and model initialization. Because the native vLLM frontend is efficiently implemented, this recurring CPU/software latency is typically about 1% of end-to-end serving latency for representative models and workloads.

4.2 rLLM Backend

As shown in Figure 2 (e), the rLLM backend evaluates workload traces received from the frontend to estimate simulated target latency, which subsequently advances the continuous simulation loop. To support multi-fidelity backend integration, we provide a unified abstraction. The abstraction allows downstream hardware simulators to implement only hardware-facing latency models while reusing the native vLLM control plane unchanged, thereby lowering the integration barrier for evaluating novel architectures.

Code 1 provides a simplified pseudocode example illustrating how an arbitrary target simulator (`SimulatorA`) is natively integrated. The new simulator simply inherits the `SimBackend` class and implements the execution logic for fundamental operators like `runLinear`, `runAttention` and `runComm`. The rLLM frontend dynamically passes a `StepContext` downward for each scheduling step. This context packages all detailed input semantics, such as exact input tensor sizes, attention metadata, collective communication shapes, and sequence lengths, providing the backend with fully sufficient information to complete its calculations independently. As a result, backend developers are completely isolated from high-level complexities such as request arrivals, continuous batching, or KV cache management. Once the operators are evaluated, per-operator simulated target latencies are aggregated and written into a `StepResult`, which is returned upward to the rLLM frontend to advance virtual time using the simulated target latency and seamlessly drive the next round of system scheduling.

This abstracted backend design seamlessly accommodates multi-fidelity hardware simulators, ranging from coarse-grained analytical models to cycle-accurate microarchitecture simulators. Across all these cases, the integration surface remains perfectly stable, making it straightforward to plug in any new hardware simulator without modifying the comprehensive serving runtime.

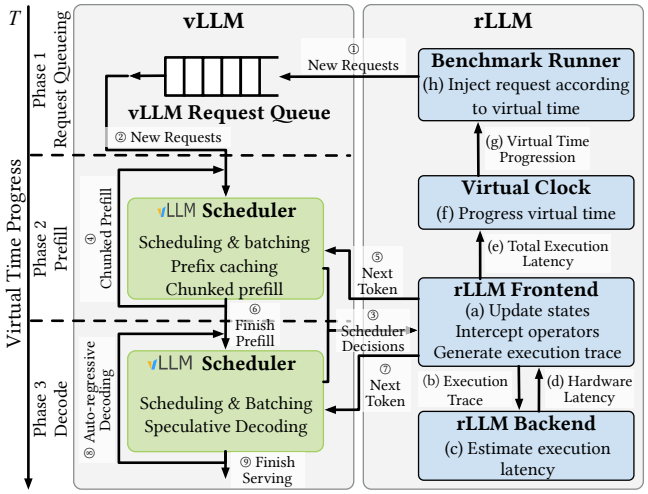


Figure 3. The request lifecycle and per-step execution flow of rLLM. The left-side timeline depicts the macroscopic request lifecycle spanning from arrival through chunked prefill to iterative decode and completion. The right-side part details the microscopic simulation loop invoked during every active execution step. This workflow effectively couples unmodified software orchestration with accurate hardware simulation.

To demonstrate this flexibility across different design trade-offs, we detail three distinct classes of backend models natively integrated into rLLM below.

Analytical Model. The most classical approach involves integrating an analytical model. This backend estimates simulated target latency for each operator based on Roofline Model [34] and calculates communication latency derived directly from data transfer volumes and peak bandwidth. This method is computationally highly efficient and is particularly suited for large-scale systems, such as supernodes, allowing developers to rapidly explore the design space while identifying the bottleneck. However, this efficiency comes at the cost of accuracy; our empirical evaluations indicate that the analytical backend exhibits a 20 to 40% error margin compared to native vLLM executions.

Profiling-based Model. Profiling-based models offer a balance between accuracy and simulation efficiency. Following established practices in recent literature [1, 6], this approach employs sampling and prediction techniques. It first performs extensive offline sampling of representative operators to build a correlation between execution latency and operator input characteristics. Subsequently, lightweight multi-layer perceptron (MLP) networks are trained to predict the simulated target latency of operators under unseen runtime inputs. This methodology is highly suitable for evaluating existing GPU clusters: a one-time offline profiling phase enables the backend to predict individual GPU-kernel latencies with under 10% operator-level error and high online execution efficiency.

This backend prediction error is distinct from the end-to-end serving error reported in Figure 4, which also reflects the propagation of latency errors through online scheduling and batching decisions. However, a limitation is its dependency on physical hardware for data collection, making it incapable of modeling emerging, exploratory architectures like Near-Memory Processing or hybrid-bonding architectures.

Microarchitecture Simulator. To validate the performance of novel AI accelerators and experimental hardware architectures, the computer architecture community has developed numerous microarchitectural simulators, ranging from centralized GPUs [18], Near-Memory-Processing units [22, 36], to wafer-scale systems [33]. These existing simulators provide robust modeling of diverse hardware features, reporting metrics including operator execution latency, collective communication delays, and even fine-grained energy and area estimates. This integration provides rLLM with comprehensive simulation capabilities across a spectrum of novel architectures, establishing a solid foundation for evaluating emerging hardware under realistic LLM serving settings.

4.3 Dynamic Benchmark Runner and Virtual Clock

Dynamic Benchmark Runner. rLLM introduces a benchmark runner to drive the native vLLM engine with realistic online arrivals, as illustrated in Figure 2 (a). Rather than reusing the full vLLM API server, we directly instantiate the engine-side benchmark path. This choice avoids the additional complexity of HTTP serving, networking, and separate process management, while preserving the core serving semantics. Crucially, our request construction, arrival scheduling, and metric definitions strictly adhere to native vLLM conventions, ensuring seamless workload alignment. Since API-server-side request handling can execute asynchronously with the engine core and is not the dominant component of model serving latency, omitting this layer does not materially distort the benchmark results we report.

Concretely, the runner accepts the same request-level inputs used by vLLM benchmarks, including prompt content, prompt length, and target output length, and converts them into a normalized request stream. It then samples request arrivals using the same controls adopted in vLLM, including request rate, burstiness, and ramp-up policy, and builds a monotonic arrival schedule for the simulated workload. During execution, arrived requests are injected into the unmodified vLLM scheduler through a lightweight engine adapter, which preserves native request admission, continuous batching, and cache-management behavior. After the run finishes, rLLM converts the completed request timelines into vLLM-compatible benchmark outputs and aggregates system metrics such as throughput, time to first token (TTFT), time per output token (TPOT), inter-token latency (ITL), and end-to-end latency under the same output schema used by vLLM.

Because rLLM is a performance simulator rather than a numerical inference engine, it does not rely on online tensor computation to determine generation results. Instead, each request carries a replayable token trace collected offline, and the frontend directly emits the next recorded token ID after each simulated decoding step until an `EOS` token or trace exhaustion is reached. In essence, our methodology completely bypasses physical tensor execution, effectively "tricking" the native vLLM engine into operating under the illusion that it is driving real hardware. This dummy-trace design allows rLLM to preserve the exact request lifetime and termination behavior without executing real GPU kernels online.

Virtual Clock. rLLM employs a global virtual clock to represent the serving time perceived by the system, as shown in Figure 2 (b). This abstraction is necessary to cleanly decouple the simulated target latency of an LLM serving step from the backend host execution time required by the backend simulator to produce that estimate.

To avoid this mismatch, rLLM advances time in an event-driven manner. The virtual clock is owned centrally by the benchmark session and progresses only in two cases. When the system is idle, it performs an *idle jump* to the next request arrival timestamp, ensuring that future requests become immediately visible. When a vLLM inference step is executed, the backend returns the simulated target latency, and rLLM advances virtual time by that latency. Request scheduling events are recorded at the step start time, while token emission and request completion are committed at the step end time. As a result, every benchmark metric is derived from a single coherent virtual time that faithfully captures the interaction between request arrivals, native vLLM scheduling decisions, and simulated target latency.

4.4 The Request Lifecycle and Per-Step Execution Flow

The execution process of rLLM can be divided into the macroscopic lifecycle of an individual request and the microscopic per-step system workflow, as shown in Figure 3.

The lifecycle of a single inference request begins when ① the dynamic benchmark runner injects it into the unmodified vLLM request queue. The vLLM scheduler subsequently ② processes the queued request according to its scheduling policy. It dynamically packs the initial prefill stage of the request alongside other ongoing prefill or decode tasks, using chunked prefill techniques. The scheduler generates a decision specifying which requests to process in each step and sends it to the rLLM frontend. Upon receiving the scheduler decisions, rLLM starts its per-step workflow. We introduce the per-step workflow in the later part of this section. The request then undergoes one or multiple ④ execution steps until its ⑥ prefill phase fully concludes. Upon finishing the final prefill chunk, the rLLM frontend retrieves the first generated token id from a pre-collected request trace. It ⑤ feeds this token back to the vLLM frontend to simulate realistic generation. Subsequently, this request transitions to the decode phase. The request then participates in continuous batching with

other concurrent sequences. The vLLM scheduler allocates additional KV cache blocks during this phase to accommodate the growing sequence length. This ⑧ iterative decode loop persists until the ⑨ request either emits an end-of-sequence token or reaches sequence length limit.

To drive the progression of these requests, rLLM executes a continuous per-step workflow. During each execution step, the vLLM scheduler ③ dispatches the scheduler result directly to the rLLM frontend. The backend dispatch includes essential per-step metadata. This metadata explicitly defines the token counts to process and the allocated KV cache block identifiers for all active sequences. The rLLM frontend subsequently (a) updates its internal device states and initiates the model forward pass. It also intercepts all performance-critical hardware primitives. This interception dynamically generates a fine-grained execution trace capturing the exact computation and memory access semantics, and the communication operations. The frontend (b) transmits this trace to the multi-fidelity simulation backend. The backend (c) evaluates the semantic operations to estimate simulated target latency. It then (d) returns the simulated target latency to the frontend. The frontend calculates the total step duration by combining the simulated target latency with the measured CPU/software latency. It (e) forwards this comprehensive timing data to the virtual clock module to (f) advance virtual time. The advancement of the virtual time is (g) sent to the benchmark runner and directly dictates the (h) volume of new requests injected into the system. This closed-loop synchronization forces the scheduler to experience realistic queuing dynamics and instantaneous request pressure under different simulated hardware architectures.

5 Simulator Fidelity Evaluation

We evaluate rLLM across two dimensions: simulation fidelity and execution efficiency. First, we quantify how accurately the simulator reproduces the end-to-end serving behavior of vLLM. Second, we measure the simulator's host wall-clock speedup relative to executing vLLM.

Models and Workloads. We evaluate the framework using three representative setups: `Llama-3.1-8B` (TP=1), `Qwen3-32B` (TP=1), and `Llama-3.1-70B` (TP=4). To cover diverse serving patterns, we test combinations of workloads and request arrival rates (2, 4, and 8 QPS) to stress-test the vLLM scheduler. As shown in Table 1, our base workloads include *ShareGPT* (conversational, long decodes) and *LongBench* (compute-bound long prefills, short decodes), together with two multi-round variants for prefix-caching ablation.

Hardware Setup. All experiments are conducted on a dual-socket server with two Intel Xeon Gold 6548Y+ CPUs (128 logical CPU cores in total) and eight NVIDIA H20 GPUs connected through a full NV18 NVLink topology.

Table 1. Representative workloads used in evaluation; (M) denotes the multi-round shared-prefix variants.

Workload	Avg. Prefill	Avg. Decode	Size	Reuse Rate
ShareGPT	1348	785	30	–
LongBench	4000	16	30	–
ShareGPT (M)	886	941	30	34.5%
LongBench (M)	4359	15	30	79.9%

Baselines. We select two state-of-the-art LLM serving simulators, LLMServingSim [5, 6] and Vidur [1] as our baselines. In our evaluation, both baselines use profiling-based hardware performance models, enabling a direct comparison with rLLM-P of how serving-stack integration affects end-to-end fidelity. To demonstrate the fidelity, we compare the simulated serving metrics with those from native vLLM execution. We use a recent vLLM version 0.16.0 in our evaluation as it is the latest version at the time of rLLM development. We align the request arrival patterns, including input length, output length, and request arrival time, and other serving setups (e.g., prefix caching), for all experiments. Within each simulator-to-vLLM comparison, we align the model, complete request trace, QPS, tensor parallelism, serving configuration, and metric definitions.

rLLM Backend. We evaluate rLLM with three backend configurations of different fidelity levels, denoted as rLLM-P, rLLM-A, and rLLM-μ throughout this paper. rLLM-P uses the profiling-based backend, which looks up offline-profiled kernel latencies and is the most accurate configuration. rLLM-A uses our self-maintained analytical model parameterized for the NVIDIA H20 GPU, estimating simulated target latency for each operator with a Roofline-style model and communication latency from data-transfer volumes and peak bandwidth. rLLM-μ uses FlashGPU-Sim [38], an open-source GPU microarchitecture simulator, as the simulation backend, where offline-generated cycle-accurate data drives runtime lookup of simulated target latency. All three backends share the same vLLM frontend and request traces, so differences in their results primarily arise from the modeled target latency and its interaction with the native serving loop rather than from different serving-stack implementations.

5.1 Evaluation Results

Accuracy. Figure 4 compares the normalized TTFT, TPOT, and throughput of all configurations and summarizes their end-to-end fidelity using MAPE and MedAPE. The results support two main observations: rLLM-P provides the highest overall fidelity, while rLLM-A and rLLM-μ show that rLLM remains accurate even with less precise hardware backends.

First, rLLM-P achieves the lowest MAPE and MedAPE across all three metrics among the evaluated configurations and baselines. Its MAPE values are 20.35%, 21.94%, and 10.16% for TTFT, TPOT, and throughput, respectively, with corresponding MedAPE values of 16.92%, 20.83%, and 9.32%. This result establishes the profiling-based configuration as the most accurate end-to-end realization of rLLM.

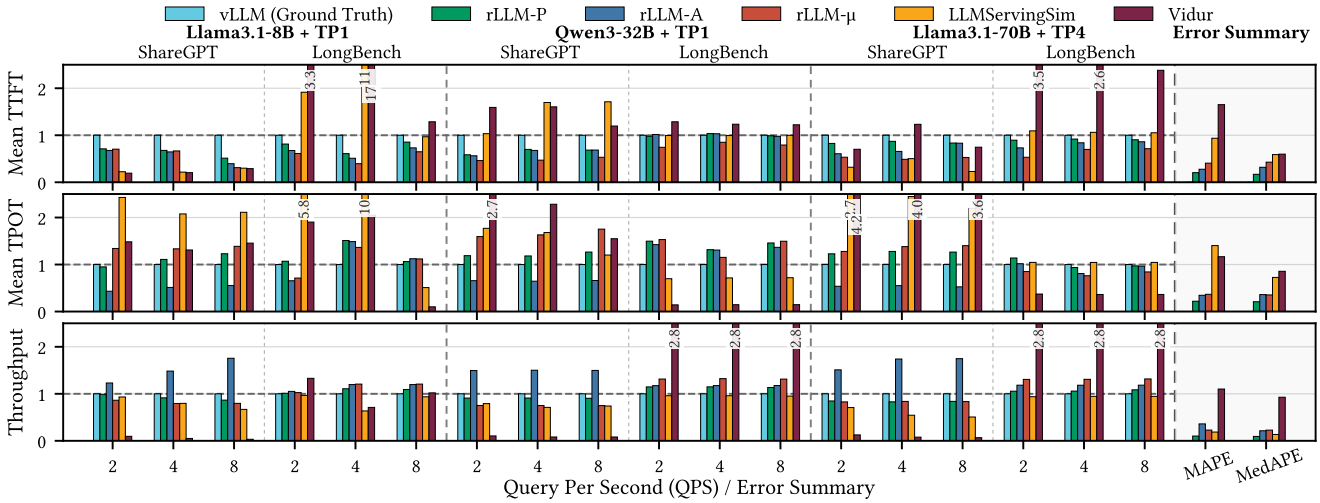


Figure 4. End-to-end performance fidelity of rLLM compared with baselines across diverse model configurations and workloads. The comparison evaluates mean TTFT, mean TPOT, and throughput relative to the vLLM execution under varying query arrival rates. Each simulator is normalized against the corresponding real-vLLM execution under matched experimental settings. rLLM is evaluated with three backend configurations: profiling-based model (rLLM-P), analytical model (rLLM-A), and microarchitecture simulator (rLLM-μ). Across all evaluated scenarios, rLLM-P achieves the lowest Mean Absolute Percentage Error (MAPE) and Median Absolute Percentage Error (MedAPE) among all evaluated configurations and baselines. This superior accuracy stems from the native integration of the vLLM orchestration stack. Our framework effectively reduces the compounding modeling errors and semantic gaps common in isolated serving simulators.

Second, the additional rLLM configurations remain competitive despite the larger errors introduced by their hardware backends. For TTFT and TPOT, rLLM-A and rLLM-μ achieve MAPE values between 27.68% and 40.65%, substantially below LLMServingSim's 93.66–140.29% and Vidur's 116.67–165.08%. For throughput, rLLM-A and rLLM-μ achieve MAPE values of 36.01% and 22.85%, respectively, both below Vidur's 110.13%, although LLMServingSim remains more accurate at 18.65%. MedAPE exhibits the same overall trend, as summarized in Figure 4. The three configurations expose different fidelity and simulation-cost trade-offs. rLLM-P provides the highest accuracy when the target hardware is available for offline profiling, while rLLM-A offers inexpensive estimation of simulated target latency suitable for large design-space exploration. Because directly invoking cycle-level simulation for complete serving workloads is prohibitively expensive, rLLM-μ uses offline FlashGPU-Sim cycle samples with runtime lookup and interpolation; limited sample coverage and residual errors in the underlying microarchitecture model may contribute to its higher end-to-end error.

Together, these results separate backend accuracy from serving-stack fidelity. By preserving native vLLM scheduling and KV cache semantics, rLLM avoids the frontend discrepancies that compound end-to-end errors in simulators with independently implemented serving stacks. Consequently, rLLM maintains strong fidelity across backend models with different accuracy and simulation-cost trade-offs.

Efficiency. For each simulated step, the profiling-based backend identifies the required kernels, looks up their offline-profiled latencies, and aggregates these latencies to determine the corresponding virtual-time advancement. This approach avoids time-consuming physical GPU execution, thereby providing substantial host wall-clock speedups. As shown in Table 2, rLLM-P achieves 10.82–21.06× speedup for `Llama-3.1-8B` with TP1, 39.20–54.68× for `Qwen3-32B` with TP1, and 8.90–11.70× for `Llama-3.1-70B` with TP4 across QPS 2, 4, and 8. The speedup increases with model size at the same parallelism because native GPU execution becomes more expensive, whereas the cost of profile lookup and latency aggregation does not increase proportionally. This efficiency makes it practical to explore inference configuration design spaces for large-scale models in laboratory environments.

rLLM's Prefix Caching. rLLM inherits vLLM's native prefix-caching path, which reuses previously computed KV cache when a newly arrived request shares a prompt prefix with earlier requests. This optimization is especially important for multi-turn conversations and agentic workflows, where many requests repeatedly traverse the same prompt context. To demonstrate that rLLM accurately preserves this performance dynamic, we structure the base ShareGPT and LongBench traces into multi-round variants, *ShareGPT (M)* and *LongBench (M)*. We define *reuse rate* as the fraction of tokens that can utilize the prefix cache. *ShareGPT (M)* and

Table 2. Benchmark-loop host wall-clock time of rLLM-P and native vLLM. Speedup is calculated as the vLLM time divided by the rLLM-P time.

Model	QPS	vLLM (s)	rLLM-P (s)	Speedup
Llama-3.1-8B (TP1)	2	71.05	6.56	10.82×
	4	65.52	4.02	16.31×
	8	64.75	3.08	21.06×
Qwen3-32B (TP1)	2	198.47	3.63	54.68×
	4	194.88	4.67	41.71×
	8	194.23	4.96	39.20×
Llama-3.1-70B (TP4)	2	166.69	18.72	8.90×
	4	165.65	14.16	11.70×
	8	164.66	15.98	10.30×

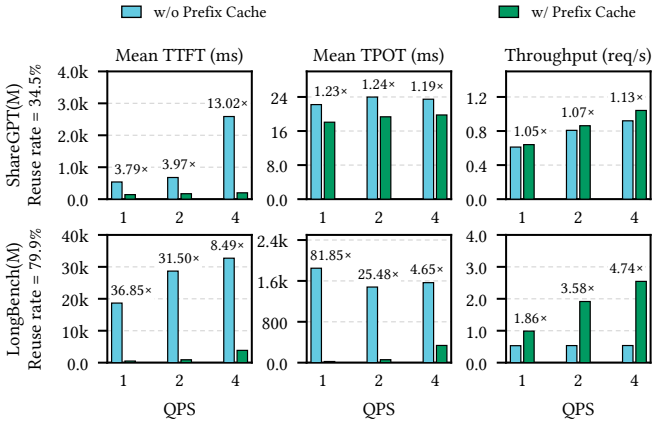


Figure 5. Prefix-caching ablation on multi-round workloads. rLLM natively supports prefix caching, a critical serving optimization for multi-turn LLM generation. Workloads with higher cache reuse rates yield significantly greater performance improvements when prefix caching is enabled.

LongBench (M) expose reuse rates of 34.5% and 79.9%, respectively. We evaluate these workloads on an analytical backend parameterized for the NVIDIA H20 GPU using `Qwen3-32B`, and compare rLLM with prefix caching enabled versus disabled at QPS $\in \{1, 2, 4\}$.

Figure 5 shows that enabling prefix caching consistently improves request throughput while reducing both mean TTFT and mean TPOT, and the gain depends on the workload reuse rate. On *ShareGPT (M)*, prefix caching increases request throughput by 4.7–13.2% across QPS 1, 2, and 4, and reduces mean TTFT by 73.6–92.3% and mean TPOT by 15.8–19.3%. The effect is substantially larger on *LongBench (M)* with much higher cache reuse rate: request throughput improves by 1.86×–4.74×, while mean TTFT and mean TPOT drop by 88.2–97.3% and 78.5–98.8%, respectively. These results show that rLLM preserves native serving-side features such as prefix caching and can expose their end-to-end impact under realistic online workloads.

6 Case Study

Emerging architectural designs targeting LLM serving often assess their performance using offline evaluation, which relies on static proxies such as fixed batch sizes and sequence lengths. While these static proxies simplify measurement, they fail to capture the dynamic nature of online serving in production environments. In practice, online serving systems employ techniques like continuous batching, which dynamically interleaves requests and constantly shifts the hardware bottleneck between compute capacity and memory bandwidth. Evaluating architectures without these system-level dynamics misestimates their true performance (a phenomenon we term *speedup inflation*) and misguides DSE toward suboptimal hardware configurations.

To demonstrate the necessity of serving-aware simulation, we conduct a case study based on two research questions:

- **Q1 (Fair Evaluation):** Do hardware speedups reported under static offline evaluation transfer to online serving? (Section 6.2)
- **Q2 (Serving-Aware DSE):** Do the optimal hardware configurations derived under offline evaluation remain optimal under online serving-aware metrics? (Section 6.3)

Addressing these questions requires system-wide evaluation that isolated architectural simulators cannot provide. By comprehensively coupling a production-grade LLM serving framework with an operator-level hardware backend, rLLM captures the exact interplay between scheduling algorithms and hardware execution, enabling a fair and rigorous evaluation environment for emerging accelerators.

6.1 Integration

Target Architecture. To answer the aforementioned research questions, we use rLLM to conduct a case study on H2-LLM [22]. H2-LLM is a state-of-the-art heterogeneous accelerator targeting LLM inference. Compared to a Centralized Processor (CP) baseline representing a conventional compute-centric architecture defined in the original study, H2-LLM mitigates the external memory bandwidth bottleneck during memory-intensive decoding via a Hybrid-Bonding-based Near-Data Processing (NDP) architecture. By placing customized processing elements directly alongside memory banks, H2-LLM harnesses high internal memory bandwidth for memory-bound operations, thereby achieving a significant performance speedup. A core contribution of H2-LLM is its DSE framework, which automatically navigates the design space to find the optimal architecture and dataflow co-design. However, like many emerging architectural proposals, it relies on a simplified offline methodology to evaluate these designs. Because it represents architectures that offer significant hardware innovation but rely on static serving assumptions, H2-LLM provides a baseline to test whether statically optimized architectures remain optimal under online serving.

System Integration. As introduced in Section 4.2, rLLM exposes an abstract `SimBackend` interface for integrating customized hardware simulators. To connect H2-LLM, we leverage the performance evaluators provided in the H2-LLM open-source artifact and construct an adapter to overload this interface. This adapter encapsulates H2-LLM's original hardware latency models and DSE metrics, enabling it to provide estimates of operator-level simulated target latency to rLLM. In this way, we preserve H2-LLM's architectural semantics but transfer the entire system control plane to the native vLLM engine. rLLM directly drives the H2-LLM backend using vLLM's native request emission, scheduling scheme, and KV cache management policy, subjecting the hardware model to the execution patterns induced by realistic online serving.

6.2 Fair Evaluation: The Speedup Inflation

Offline vs. Online Evaluation. To answer Q1, our first experiment examines whether hardware speedups reported under offline evaluation transfer to real-world serving. Offline evaluation (as used in the original H2-LLM paper) characterizes performance at fixed batch sizes and sequence lengths to approximate serving traffic intensity. This methodology relies on static proxies, inherently modeling requests as arriving simultaneously and sharing identical request lengths. In contrast, online evaluation captures the dynamic nature of production, where requests arrive continuously over time (e.g., following a Poisson process parameterized by QPS) and feature varying and heterogeneous prefill and decode lengths. Online evaluation reflects the true hardware utilization and performance under practical serving constraints.

Evaluation Methodology. To examine whether the significant speedup over CP reported in the H2-LLM paper holds in deployed environments, we evaluate the speedup of H2-LLM relative to the CP baseline under both offline and online evaluation modes. We adopt the exact optimized hardware configurations used for the H2-LLM and CP comparison in their offline evaluation, and apply the same configurations within rLLM's online serving loop. Consistent with the H2-LLM paper, we use the Llama-3.1-8B model and the *ShareGPT* dataset (detailed in Table 1). We instantiate the optimal hardware designs found for offline batch sizes 1, 4, and 16 (as the original paper did), and measure their online end-to-end latencies under a range of QPS levels in rLLM.

Results. The results in Figure 6 reveal two distinct insights. First, mean end-to-end latency is predictably higher in online evaluation because requests accumulate queueing delays prior to execution, which static models omit. Second, offline evaluation overestimates the acceleration provided by NDP, resulting in substantial speedup inflation. For example, under static offline assumptions, H2-LLM reports 31.0× and 24.8× speedups over the CP baseline for batch sizes 1 and 4. However, when evaluated online at a moderate load (QPS = 4), the speedup for the batch-size-1 optimized design drops to 6.8×, a factor of 4.5× smaller than predicted. This inflation exacerbates as traffic intensity (QPS) increases.

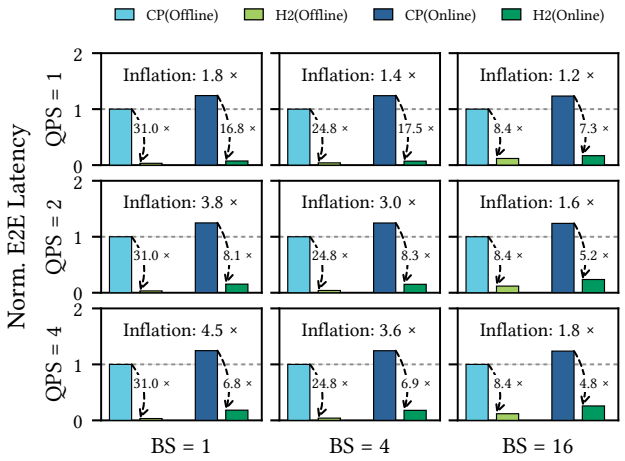


Figure 6. Speedup of H2-LLM over the Centralized Processor (CP) baseline with and without rLLM dynamism. The speedup inflation is quantified as the ratio of the offline speedup to the online speedup. The results show that static evaluation overestimates speedups.

This discrepancy stems from the fundamental shift in operational intensity induced by the online serving loop. Offline evaluation assumes a homogeneous workload where compute-bound prefill phases and memory-bound decode phases are strictly separated, analogous to early static batching methodologies. However, continuous batching dynamically interleaves compute-intensive chunked prefill iterations with memory-intensive decode iterations within the same scheduling step. This injection of compute-heavy work structurally increases the overall compute-to-bandwidth ratio, alleviating the CP's bandwidth bottleneck and proportionately eroding the advantage of the bandwidth-specialized NDP architecture.

6.3 Serving-Aware DSE

To address Q2, we investigate whether the optimal hardware configurations derived under offline DSE remain optimal when evaluated against online serving-aware metrics. We first revisit the original DSE framework proposed in H2-LLM, which employs a genetic algorithm to iteratively narrow the hardware search space to find the optimal architecture-dataflow co-design. However, H2-LLM's methodology fundamentally relies on offline evaluation: its optimization objective is to minimize the latency under fixed batch size. To obtain the baseline configuration, we replicate H2-LLM's original offline DSE over the same search space. As detailed in Table 3, the search space includes Batch Size, Near-Memory Processing (NMP) channels, FPU count, FPU frequency, and memory bandwidth. Crucially, as highlighted in the H2-LLM study, allocating resources forces a rigid trade-off between compute density (FPU count and frequency) and memory bandwidth due to strict Area and Thermal Design Power

Table 3. Design Space Exploration Parameters

Design Parameter	Exploration Space
Batch Size	{1, 2, 4, 8}
NMP Channels	{2, 4, 6, 8}
FPUs per PE	{1, 2, 4, 8}
FPU Frequency (MHz)	{400, 600, 800, 1000}
Memory Bandwidth (GB/s)	{12.8, 25.6, 51.2}

(TDP) constraints: aggressive investments in bandwidth necessitate stark reductions in compute capacity.

To investigate how architectures optimized under offline settings actually performed realistically, we shift our evaluation target to online serving metrics. Following standards adopted by the LLM system community, our evaluation metric maximizes sustainable request throughput under strict Service-Level Objective (SLO) constraints. This aligns with the goal of maximizing the number of concurrently served users while strictly satisfying latency SLOs. Through rLLM, we migrate the optimal configuration derived from the offline DSE into the online serving loop and measure its true serving throughput alongside critical tail latencies.

In comparison to offline DSE, our proposed *Serving-Aware DSE* aims directly at finding the hardware limits that maximize throughput under these strict SLO constraints. The DSE algorithm remains identical to the one employed in H2-LLM. The fundamental shift lies exclusively in replacing the simplistic offline latency estimator with rLLM's end-to-end continuous serving loop evaluator, thereby integrating serving dynamics directly into the architectural optimization loop.

DSE evaluation cost. Following the genetic-algorithm-based DSE workflow of H2-LLM, we reduce the nominal 768-point design space to 25 promising candidates for system-level evaluation. With rLLM's efficient simulation speed, evaluating each candidate takes only 9.8 s on average, allowing a single worker to complete the DSE for one QPS setting in approximately 4.1 min.

Results. We conduct the DSE using the *ShareGPT* dataset under stringent SLOs: a P95 TTFT $\leq$ 400 ms and a P95 TPOT $\leq$ 40 ms. We evaluate the optimal architectures across varying traffic intensities (QPS 2, 4, and 8). The results in Figure 7 demonstrate that targeting online serving metrics systematically shifts the Pareto frontier. By adopting the serving-aware DSE, the system unlocks up to a 1.32× enhancement in sustainable request throughput compared to the offline DSE optimal. Concurrently, these configurations materially reduce the mean end-to-end latencies by over 30% at identical traffic loads.

To detail the hardware changes, Table 4 shows the precise configurations selected across varying QPS levels. Offline DSE consistently favors a bandwidth-heavy configuration (51.2 GB/s bandwidth, 4 FPUs/PE, 800 MHz). In contrast, the serving-aware DSE systematically selects compute-dominant architectures. For instance, at moderate-to-heavy

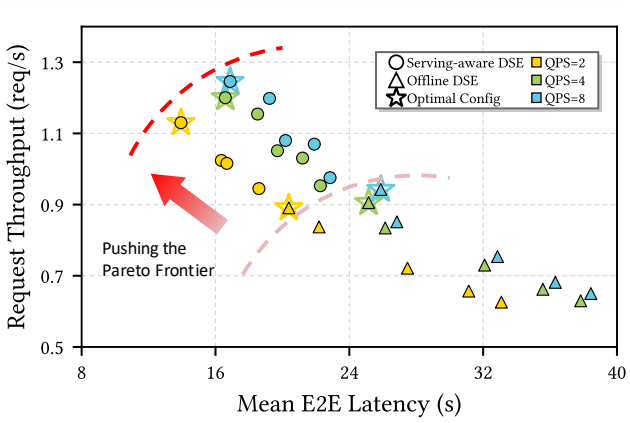


Figure 7. Comparison of offline and serving-aware DSE under realistic online serving conditions. Enabled by rLLM's co-simulation framework, serving-aware DSE identifies hardware configurations that achieve both higher request throughput and lower end-to-end latency across QPS levels, shifting the Pareto frontier beyond the offline baseline.

Table 4. Comparison of optimal hardware configurations selected by offline versus serving-aware DSE across varying traffic loads (differences highlighted in bold). The serving-aware framework systematically prioritizes compute capacity over memory bandwidth to accommodate the mixed-batching dynamics of online serving.

QPS	Target	#FPU	Frequency	Bandwidth
2	Offline	4	**800**	**51.2**
	Serving-aware	**8**	**600**	**25.6**
4	Offline	4	800	**51.2**
	Serving-aware	**8**	800	**12.8**
8	Offline	4	800	**51.2**
	Serving-aware	**8**	800	**12.8**

loads (QPS $\geq$ 4), the optimal configuration substantially reduces bandwidth to 12.8 GB/s while doubling compute capacity to 8 FPUs/PE.

This drift occurs because offline DSE over-optimizes for the memory-bound nature of isolated decoding. However, the interleaving of prefill and decode caused by continuous batching increases the overall compute intensity of the workload. Our choice effectively trades excessive memory bandwidth for computational density, improving overall performance.

6.4 Key Insights

Our case study on H2-LLM reveals two key limitations of conventional offline architectural analysis. First, offline evaluation tends to overestimate hardware gains because it ignores online serving effects such as continuous batching, which changes the compute-to-bandwidth ratio seen by the hardware and prevents architectural speedups from translating directly into end-to-end system improvements. Second, hardware configurations that appear optimal under static

assumptions often become suboptimal under practical SLO-constrained serving, where mixed prefill and decode execution shifts the design balance from pure bandwidth optimization toward greater compute capacity. Together, these results highlight the need for serving-aware co-simulation frameworks such as rLLM to obtain realistic performance comparisons and sound design conclusions.

7 Conclusion

We presented rLLM, a serving-aware co-simulation framework that couples the native vLLM serving stack with pluggable hardware simulators through a plugin interface. Our evaluation across profiling-based, analytical, and microarchitecture backends demonstrates strong end-to-end fidelity across different backend accuracy and simulation-cost tradeoffs. Our results show that realistic serving dynamics materially change both performance outcomes and architectural design conclusions, making conventional static evaluation insufficient for LLM accelerator studies. rLLM provides a practical foundation for future serving-aware LLM hardware co-design.

Acknowledgments

This research was partially conducted by ACCESS AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government. It was also supported partially by Research Grants Council of Hong Kong SAR (T41-517/25-N & STG3/E-605/25-N & 16213824 & 16212825 & T46-415_25-R).

References

[1] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A large-scale simulation framework for llm inference. *Machine Learning and Systems (MLSys)* (2024).

[2] Amey Agrawal, Mayank Yadav, Sukrit Kumar, Anirudha Agrawal, Garv Ghai, Souradeep Bera, Elton Pinto, Sirish Gambhira, Mohammad Adain, Kasra Sohrab, et al. 2026. Revati: Transparent GPU-Free Time-Warp Emulation for LLM Serving. *arXiv preprint arXiv:2601.00397* (2026).

[3] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 163–174.

[4] Yuzong Chen, Chao Fang, Xilai Dai, Yuheng Wu, Thierry Tambe, Marian Verhelst, and Mohamed S. Abdelfattah. 2026. P^3-LLM: An Integrated NPU-PIM Accelerator for Edge LLM Inference Using Hybrid Numerical Formats. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. 604–619. doi:10.1109/ISCA66397.2026.00055

[5] Jaehong Cho, Hyunmin Choi, and Jongse Park. 2025. LLMServingSim2.0: A Unified Simulator for Heterogeneous Hardware and Serving Techniques in LLM Infrastructure. *IEEE Computer Architecture Letters (CAL)* 24, 2 (2025), 361–364. doi:10.1109/LCA.2025.3628325

[6] Jaehong Cho, Minsu Kim, Hyunmin Choi, Guseul Heo, and Jongse Park. 2024. LLMServingSim: A HW/SW Co-Simulation Infrastructure for LLM Inference Serving at Scale. In *IEEE International Symposium on Workload Characterization (IISWC)*. 15–29. doi:10.1109/IISWC63097.2024.00012

[7] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv preprint arXiv:2307.08691. https://arxiv.org/abs/2307.08691

[8] Deloitte. 2025. TMT Predictions 2026: The AI Gap Narrows but Persists. https://www.deloitte.com/us/en/insights/industry/technology/technology-media-and-telecom-predictions.html Accessed: 2026-04-12.

[9] Deloitte. 2025. Why AI's Next Phase Will Likely Demand More Computational Power, Not Less. https://www.deloitte.com/us/en/insights/industry/technology/technology-media-and-telecom-predictions/2026/compute-power-ai.html Accessed: 2026-04-12.

[10] Cenlin Duan, Jianlei Yang, Rubing Yang, Yikun Wang, Yiou Wang, Lingkun Long, Yingjie Qi, Xiaolin He, Ao Zhou, Xueyan Wang, and Weisheng Zhao. 2026. HPIM: Heterogeneous Processing-In-Memory-based Accelerator for Large Language Models Inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2026). doi:10.1109/TCAD.2026.3710643

[11] Zizhuo Fu, Xiaotian Guo, Wenxuan Zeng, Shuzhang Zhong, Yadong Zhang, Peiyu Chen, Runsheng Wang, Le Ye, and Meng Li. 2025. H2EAL: Hybrid-Bonding Architecture with Hybrid Sparse Attention for Efficient Long-Context LLM Inference. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–9. doi:10.1109/ICCAD66269.2025.11240713

[12] Gartner. 2025. Gartner Says AI-Optimized IaaS Is Poised to Become the Next Growth Engine for AI Infrastructure. https://www.gartner.com/en/newsroom/press-releases/2025-10-15-gartner-says-artificial-intelligence-optimized-iaas-is-poised-to-become-the-next-growth-engine-for-artificial-intelligence-infrastructure Accessed: 2026-04-12.

[13] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. 2025. PIM Is All You Need: A CXL-Enabled GPU-Free System for Large Language Model Inference. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 862–881. doi:10.1145/3676641.3716267

[14] Siyuan He, Peiran Yan, Yandong He, Youwei Zhuo, and Tianyu Jia. 2025. Tasa: Thermal-aware 3D-Stacked Architecture Design with Bandwidth Sharing for LLM Inference. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–9. doi:10.1109/ICCAD66269.2025.11240805

[15] Yintao He, Haiyu Mao, Christina Giannoula, Mohammad Sadrosadati, Juan Gómez-Luna, Huawei Li, Xiaowei Li, Ying Wang, and Onur Mutlu. 2025. PAPI: Exploiting Dynamic Parallelism in Large Language Model Decoding with a Processing-In-Memory-Enabled Computing System. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 766–782. doi:10.1145/3676641.3716009

[16] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 722–737. doi:10.1145/3620666.3651380

[17] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i: Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*. IEEE, 1–14. doi:10.1109/ISCA52012.2021.00010

[18] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*.

[19] KyungSoo Kim, Omin Kwon, Yeonhong Park, and Jae W. Lee. 2025. AiDE: Attention-FFN Disaggregated Execution for Cost-Effective LLM Decoding on CXL-PNM. *IEEE Computer Architecture Letters (CAL)* 24, 2 (2025), 285–288. doi:10.1109/LCA.2025.3597323

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[21] Manuel Le Gallo. 2025. In-memory Computing Approaches for Large Language Model Acceleration for IEDM 2025. https://research.ibm.com/publications/in-memory-computing-approaches-for-large-language-model-acceleration IBM Research publication page. Accessed: 2026-04-12.

[22] Cong Li, Yihan Yin, Xintong Wu, Jingchen Zhu, Zhutianya Gao, Dimin Niu, Qiang Wu, Xin Si, Yuan Xie, Chen Zhang, and Guangyu Sun. 2025. H2-LLM: Hardware-Dataflow Co-Exploration for Heterogeneous Hybrid-Bonding-based Low-Batch LLM Inference. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. 194–210. doi:10.1145/3695053.3731008

[23] Cong Li, Zhe Zhou, Size Zheng, Jiaxi Zhang, Yun Liang, and Guangyu Sun. 2024. SpecPIM: Accelerating Speculative Inference on PIM-Enabled System via Architecture-Dataflow Co-Exploration. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 950–965. doi:10.1145/3620666.3651352

[24] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. In *IEEE Computer Architecture Letters*, Vol. 19. IEEE, 106–109. doi:10.1109/LCA.2020.2973991

[25] McKinsey & Company. 2025. The Next Big Shifts in AI Workloads and Hyperscaler Strategies. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-next-big-shifts-in-ai-workloads-and-hyperscaler-strategies Accessed: 2026-04-12.

[26] Yue Pan, Zihan Xia, Po-Kai Hsu, Lanxiang Hu, Hyungyo Kim, Janak Sharda, Minxuan Zhou, Nam Sung Kim, Shimeng Yu, Tajana Rosing, and Mingu Kang. 2025. Stratum: System-Hardware Co-Design with Tiered Monolithic 3D-Stackable DRAM for Efficient MoE Serving. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, New York, NY, USA, 1–17. doi:10.1145/3725843.3756043

[27] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. 2024. 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 (ASPLOS)*.

[28] Sang-Soo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon, Jinhyun Kim, Jieun Lee, YeonGon Cho, Yongmin Tai, Jeonghyeon Cho, Hoyoung Song, Jung Ho Ahn, and Nam Sung Kim. 2024. An LPDDR-based CXL-PNM Platform for TCO-efficient Inference of Transformer-based Large Language Models. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Los Alamitos, CA, USA, 970–982. doi:10.1109/HPCA57654.2024.00078

[29] Andrea Renda, Andrew W. Wyckoff, Brooke Tanner, Cameron F. Kerry, Derek Belle, and Elham Tabassi. 2026. Global Energy Demands Within the AI Regulatory Landscape. https://www.brookings.edu/articles/global-energy-demands-within-the-ai-regulatory-landscape/ Accessed: 2026-04-12.

[30] Minseok Seo, Jungi Hyun, Seongho Jeong, Xuan Truong Nguyen, Hyuk-Jae Lee, and Hyokeun Lee. 2025. OASIS: Outlier-Aware KV Cache Clustering for Scaling LLM Inference in CXL Memory Systems. *IEEE Computer Architecture Letters (CAL)* 24, 1 (2025), 165–168. doi:10.1109/LCA.2025.3567844

[31] Arman Shehabi, Sarah J. Smith, Alex Hubbard, Alex Newkirk, Nuoa Lei, Md Abu Bakar Siddik, Billie Holecek, Jonathan Koomey, Eric Masanet, and Dale Sartor. 2024. *2024 United States Data Center Energy Usage Report*. Technical Report LBNL-2001637. Lawrence Berkeley National Laboratory. https://eta-publications.lbl.gov/sites/default/files/2024-12/lbnl-2024-united-states-data-center-energy-usage-report_1.pdf Accessed: 2026-04-12.

[32] Taiwan Semiconductor Manufacturing Company. 2025. TSMC 3DFabric ® for High-Performance Computing. https://www.tsmc.com/english/dedicatedFoundry/technology/platform_HPC_tech_WLSI Accessed: 2026-04-12.

[33] Chris Thames, Hang Yan, and Yifan Sun. 2022. Understanding Wafer-Scale GPU Performance using an Architectural Simulator. In *Proceedings of the 14th Workshop on General Purpose Processing Using GPU (GPGPU)*. 1–2. doi:10.1145/3530390.3532736

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

[35] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. Astra-sim2. 0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[36] Tongxin Xie, Zhenhua Zhu, Bing Li, Yukai He, Cong Li, Guangyu Sun, Huazhong Yang, Yuan Xie, and Yu Wang. 2025. UniNDP: A Unified Compilation and Simulation Tool for Near DRAM Processing Architectures. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 624–640. doi:10.1109/HPCA61900.2025.00054

[37] Ceyu Xu, Yongji Wu, Xinyu Yang, Beidi Chen, Matthew Lentz, Danyang Zhuo, and Lisa Wu Wills. 2025. LLM.265: Video Codecs are Secretly Tensor Codecs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 445–460. doi:10.1145/3725843.3756078

[38] Siying Yu, Yixun Hong, Guozhi Qiu, Feng Gu, Chenbo Geng, Zhengrong Wang, Chen Zhang, and Bei Yu. 2026. FlashGPU-sim: Enabling GPU Modeling for Modern Architectures and AI Workloads. In *IEEE/ACM 59th International Symposium on Microarchitecture (MICRO)*.

[39] Hengrui Zhang, August Ning, Rohan Baskar Prabhakar, and David Wentzlaff. 2024. Llmcompass: Enabling efficient hardware design for large language model inference. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*.

[40] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.