

SimServing: Native-Execution Simulation for Evolution-Resilient LLM Serving Configuration Tuning

Yujie Huang

The Chinese University of Hong Kong
Hong Kong SAR, China
yjhuang@cse.cuhk.edu.hk

Zhihan Jiang

The Chinese University of Hong Kong
Hong Kong SAR, China
zhjiang22@cse.cuhk.edu.hk

Jiazhen Gu

The Chinese University of Hong Kong
Hong Kong SAR, China
jiazhengu@cuhk.edu.hk

Michael R. Lyu

The Chinese University of Hong Kong
Hong Kong SAR, China
lyu@cse.cuhk.edu.hk

Abstract

Large Language Model (LLM) services expose a large configuration space that critically affects performance, yet tuning these configurations on hardware (e.g., NVIDIA GPUs) is prohibitively expensive. Simulation offers a scalable alternative, but state-of-the-art simulators suffer from a fundamental software engineering pitfall: they re-implement the serving framework’s complex control logic (e.g., request scheduling and memory management), creating a semantic copy that inevitably diverges from the rapidly evolving upstream software. Our empirical study across eight consecutive vLLM versions confirms this fidelity drift, with prediction errors widening as the framework evolves.

We observe that the serving management software dominates performance fidelity yet accounts for less than 10% of total execution time, with GPU computation consuming over 90%. This asymmetry motivates SIMSERVING, a hybrid simulation framework that executes the native, unmodified serving software while replacing GPU computation with a pluggable latency estimator. This design preserves the serving management software’s control flow by executing it rather than modeling it, thereby automatically inheriting upstream updates and substantially reducing the maintenance burden associated with re-implementation. Integrating SIMSERVING with Particle Swarm Optimization for configuration tuning, our evaluation shows that it reduces throughput prediction error from 13.8% to 4.9%, achieves up to 21.18× simulation speedup, and discovers configurations that improve throughput by up to 10.2% over existing baselines.

CCS Concepts

• **Software and its engineering** → **Software performance.**

Keywords

LLM serving, simulation fidelity, software configuration tuning, native execution



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
<https://doi.org/10.1145/3832783.3834362>

ACM Reference Format:

Yujie Huang, Jiazhen Gu, Zhihan Jiang, and Michael R. Lyu. 2026. SimServing: Native-Execution Simulation for Evolution-Resilient LLM Serving Configuration Tuning. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3832783.3834362>

1 Introduction

Large Language Models (LLMs) are now widely used in modern AI applications [34, 42, 51], spanning online chat [3, 18, 37], code generation [4, 25], document summarization [13], and many SE tasks [14]. Different LLM services exhibit diverse workload characteristics and impose varying Service Level Objectives (SLOs) on key latency metrics, e.g., Time to First Token (TTFT, measuring responsiveness) and Time Per Output Token (TPOT, measuring generation smoothness), while maximizing overall throughput. To meet these requirements, modern deployments rely on complex serving frameworks (e.g., vLLM [29], MindIE [23]) that expose a large configuration space governing request scheduling, dynamic batching, and memory management [29, 50]. Serving performance depends heavily on these configuration choices, yet they are not stable engineering knobs: as frameworks evolve, changes in scheduling and memory-management mechanisms alter the performance impact of existing configurations, making a configuration that works well for one version suboptimal after an upstream update. Configuration choices can cause up to 67% variation in TTFT (Section 3), making configuration tuning a recurring and challenging task in LLM service engineering.

Tuning these configurations in a production environment is prohibitively expensive [52, 56]: LLMs necessitate high-end hardware (e.g., GPU/NPU clusters¹), and the combinatorial explosion of the parameter space makes exhaustive on-device benchmarking impractical. Consequently, researchers have turned to simulation-based approaches to predict performance and guide tuning without incurring hardware costs.

However, existing LLM serving simulators suffer from a fundamental *re-implementation pitfall*: they attempt to rewrite the serving management software’s complex control logic within the

¹For brevity, we use “GPU” throughout to denote accelerators, including both GPUs and NPUs, unless a specific platform is discussed.

simulator [1]. This design creates two critical software engineering challenges:

- **The Semantic Gap:** The control logic of modern LLM serving software is intricate and stateful. For instance, vLLM’s scheduler maintains complex state machines for request preemption, while its memory manager implements sophisticated KV cache eviction policies based on runtime memory pressure. Re-implementing this logic in a simulator inevitably introduces subtle discrepancies, such as incorrect preemption timing or mismatched eviction decisions, that lead to compounding prediction errors.
- **The Maintenance Burden:** Serving software evolves rapidly. vLLM, for example, has released eight major versions in the past year, each introducing significant changes to scheduling and memory logic. A simulator relying on re-implemented logic requires constant, labor-intensive updates to remain synchronized. In practice, simulators often fail to keep pace. For example, Vidur’s serving logic has remained largely unchanged since its release, resulting in growing fidelity gaps with each upstream update.

To quantify these challenges, we conducted two empirical studies on vLLM and MindIE (Section 3). Our first study reveals that re-implementation-based simulators suffer from *fidelity drift*: Vidur’s prediction errors widen progressively across eight consecutive vLLM versions (v0.6 to v0.13) as its hard-coded logic falls behind upstream changes. Our second study highlights a key asymmetry: while serving configuration choices cause up to 67% variation in TTFT, the serving management software itself accounts for less than 10% of total execution time, with over 90% consumed by GPU computation.

These findings motivate our core design principle: execute the serving management software natively to preserve fidelity, while approximating GPU execution with a lightweight estimator. Guided by this principle, we propose SIMSERVING, a hybrid simulation framework that inverts the traditional simulator design. Instead of re-implementing serving logic, SIMSERVING runs the unmodified source code of the production software framework on CPU, intercepting only the model-execution entry point to replace GPU forward passes with a pluggable latency estimator. This “software-in-the-loop” design preserves the scheduling and memory management fidelity for control, automatically inherits upstream updates, and supports adaptation to new hardware through re-profiling rather than code changes.

Building on this framework, we design an automated configuration tuning pipeline to efficiently search the vast configuration space. The pipeline employs Particle Swarm Optimization (PSO) [28], a population-based algorithm that generates candidate configurations and evaluates them concurrently using our fast simulator. A fitness function balances throughput maximization with SLO constraints on latency metrics (e.g., TTFT and TPOT), transforming the traditionally hardware-bound tuning process into a fast simulation-based optimization.

We evaluate SIMSERVING on two models (Llama-3.1-8B [19] and Qwen3-30B-A3B [49]) across three workloads representing diverse batch patterns: LMSYS-Chat [53] (mixed batches), ArXiv-Summarization [13] (prefill-heavy), and BWB [26] (decode-heavy).

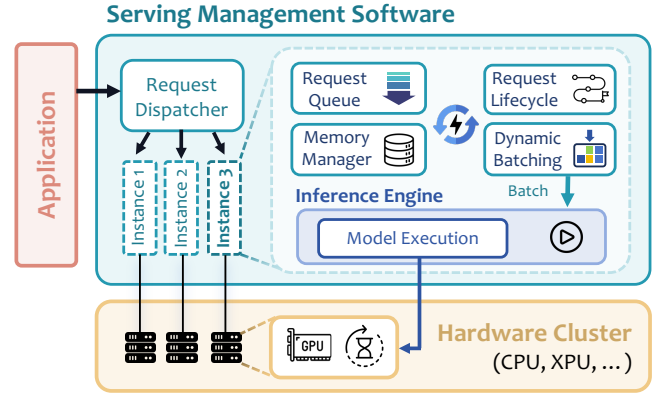


Figure 1: LLM service architecture: the serving management software orchestrates request handling and invokes the model inference engine.

Our evaluation yields three key results. First, SIMSERVING is accurate: it reduces average throughput prediction error from 13.8% (Vidur) to 4.9%, and the version-drift baseline further shows why re-implementation-based simulators become unreliable as the serving software evolves. Second, the framework is efficient: it provides simulation speedups of up to 21.18× compared to native benchmarking, with our XGBoost-based latency estimator achieving a low 6.25% average MAPE. Third, our PSO-based pipeline is effective: it discovers configurations that achieve up to 10.2% higher throughput than Vidur-Search while meeting or closely approaching the target SLO constraints, converging within 50 iterations.

In summary, this paper makes the following contributions:

- We identify and empirically characterize the re-implementation pitfall in existing LLM serving simulators, demonstrating that re-implemented logic suffers from progressive fidelity drift while the serving management software dominates performance fidelity despite negligible execution overhead.
- We propose SIMSERVING, a hybrid simulation framework that executes native serving software with a pluggable latency estimator, eliminating the semantic gap and maintenance burden of re-implementation.
- We demonstrate the practical impact of high-fidelity simulation by integrating SIMSERVING into a configuration tuning pipeline, showing that improved simulation fidelity directly translates to better tuning outcomes, with throughput gains of up to 10.2% over existing baselines.

2 Background

2.1 LLM Serving Software Stack

LLM serving is managed by a complex software stack designed to orchestrate resource-intensive inference tasks. As illustrated in Figure 1, this stack is implemented by serving frameworks such as vLLM [29], MindIE [23] and SGLang [54], and typically consists of two distinct components: the serving management software and the model inference engine.

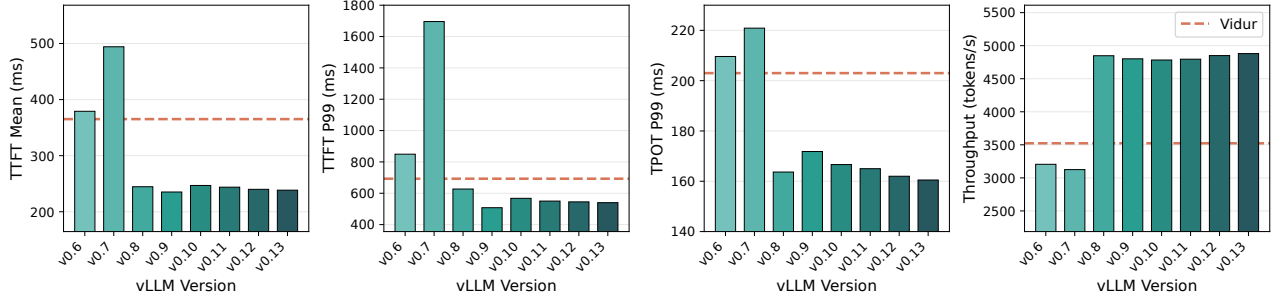


Figure 2: Simulator prediction error increases as the serving framework evolves. Vidur’s predictions diverge progressively from actual vLLM performance across versions. (Llama-3.1-8B)

Serving Management Software. This component can be viewed as the “control plane” of LLM service. It handles the complex, stateful control logic that determines how requests flow through the system, including request scheduling, dynamic batching, and KV-cache management. These decisions depend on runtime factors such as arrival patterns, queue states, memory pressure, and service-level constraints, so the behavior of this component evolves continuously as requests are admitted, grouped, preempted, and completed. The serving management software is purely software-defined and executes on the CPU. Its behavior is deterministic given a sequence of inputs, but its state space is vast due to the interplay between concurrent requests.

Model Inference Engine. This component can be viewed as the “data plane” of LLM service. It executes the tensor operations of the model on hardware accelerators like GPUs, including attention, MLP, normalization, and communication kernels in distributed settings. Unlike the serving management software, its behavior is dominated by numerical computation and hardware execution, so its latency is heavily affected by input shapes, parallelism strategy, kernel implementation, and device characteristics. Modern optimizations such as kernel fusion, custom attention kernels, quantization, and disaggregated or multi-stage execution can substantially improve efficiency, but they still operate within a performance space that is more regular and easier to approximate than the stateful control logic in the serving management software, an asymmetry we quantify in Section 3.2.

2.2 Performance Metrics and SLOs

To evaluate the quality of the LLM service, three key metrics are universally adopted. These metrics often form the Service Level Objectives (SLOs) that the service must meet:

- **Time to First Token (TTFT):** The latency from the user sending a request to receiving the first generated token.
- **Time Per Output Token (TPOT):** The average time interval between subsequent tokens.
- **Throughput:** The number of tokens generated per second across all users (Tokens/s, TPS).

There is an inherent trade-off between these metrics. For example, increasing the batch size typically improves Throughput but degrades TTFT and TPOT due to increased queuing and computation delays. Navigating this trade-off requires careful configuration

of the serving management software, which we examine in the following section.

3 Motivation

Due to the expensive hardware requirements, current configuration tuning for LLM typically requires a simulator that faithfully captures how serving software responds to different parameter settings. Several simulators have been proposed to address this need: Vidur [1] and APEX [31] model inference systems through component-level prediction and scheduling simulation, while LLM-ServingSim series [11, 12] and Frontier [15] employ analytical or cycle-accurate simulation. These simulators share a common design: they re-implement the serving management software’s control logic within the simulator. To understand the limitations of this approach, we conducted two empirical studies using vLLM and MindIE.

3.1 Study 1: Fidelity Drift Under Software Evolution

As LLM serving software rapidly evolves, a critical question is whether existing simulators can maintain their fidelity across software versions. To investigate this question, we compared Vidur’s predictions against the measured performance of eight consecutive vLLM versions (v0.6 to v0.13). Across all experiments, we varied only the target vLLM version, while keeping the model, hardware, and surrounding software stack fixed. This controlled setup allows us to attribute systematic changes in prediction error primarily to the evolution of vLLM’s serving logic rather than to differences in the experimental configuration.

As shown in Figure 2, Vidur remains relatively accurate for some earlier vLLM versions, but its predictions increasingly diverge from the measured performance as vLLM evolves. This divergence is reflected in both TTFT and TPOT:

- **Diverging Fidelity:** For v0.7, the measured mean TTFT increases sharply to nearly 500 ms, whereas Vidur continues to predict approximately 365 ms. A similar divergence appears in P99 TPOT: Vidur’s prediction remains close to 200 ms, while the measured value decreases to approximately 164 ms in later versions. These results indicate that a simulator calibrated to an earlier implementation can become progressively less representative as the underlying system evolves.

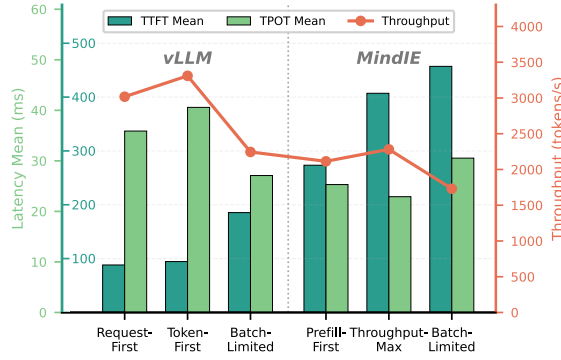


Figure 3: Serving configuration impact across vLLM and MindIE. Mean TTFT varies by up to 67% while model execution remains identical. (Llama-3.1-8B)

- **The Maintenance Gap:** This drift arises because re-implementation-based simulators encode a largely static approximation of the serving logic rather than executing the evolving upstream implementation. For example, vLLM introduced chunked-prefill scheduling in v0.5, fundamentally changing how requests are scheduled and batched. Such changes are not automatically reflected in a simulator whose hard-coded logic follows an earlier serving implementation. Keeping pace therefore requires continuous and potentially error-prone manual updates. Vidur’s revision history illustrates this cost: its scheduler remains based on vLLM V0 semantics, while an approximately 600-line port to V1 was later reverted because of instability.

As the implementation gap widens, a re-implementation-based simulator may become increasingly unsuitable for tuning modern deployments. This maintainability limitation motivates a natural alternative: executing the native serving software directly. However, doing so raises an important practical question—whether native execution is computationally feasible. We investigate this question in Study 2.

3.2 Study 2: The Case for Native Execution

To validate the feasibility of native execution, we must address two questions. First, does the serving-management software matter enough to warrant native execution? Native execution is only necessary if serving-management behavior contributes substantially to end-to-end performance. Second, is executing the native software computationally expensive? If the serving logic itself is costly, native execution would negate the speed benefits of simulation.

Impact of Software Configuration. We measured the impact of three representative serving configurations on performance for both vLLM and MindIE. As shown in Figure 3, changing the software configuration results in substantial performance variations. For instance, in MindIE, switching from Batch-Limited to Prefill-First reduces the mean TTFT from 457.1 ms to 273.5 ms, even though the underlying model execution remains identical. This confirms that the service management software logic is the primary determinant of responsiveness (TTFT) and smoothness (TPOT). A simulator must capture these logic variations with high precision to be useful.

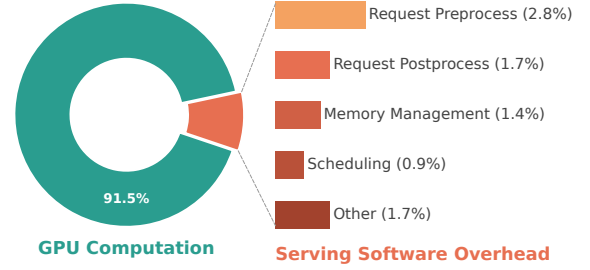


Figure 4: Serving Software overhead. Serving management software accounts for less than 10% of total overhead.

Cost of Native Execution. We then conducted an extensive temporal profiling of the critical processes of the LLM Service while serving the Qwen3-30B-A3B model [49]. Figure 4 illustrates the breakdown of the total execution cost. Surprisingly, the complex management logic (including scheduling, memory management, request pre-processing and post-processing) accounts for less than 10% of the total time. Conversely, the vast majority of the overhead, exceeding 90%, is consumed by GPU kernel execution.

Implication: This finding provides the key insight for our approach. Since the software logic is high-impact but low-cost, we should not simulate it. Instead, we should adopt a **Native Execution Strategy**. This design avoids the maintenance pitfalls of Study 1 while leveraging the cost asymmetry revealed in Study 2. We describe our realization of this principle in the next section.

4 SIMSERVING

Building on the insights from Section 3, SIMSERVING decouples the model inference engine from the serving management software, executing the latter natively while approximating the former with a pluggable latency estimator. As shown in Figure 5, the framework consists of three main components: (1) a hybrid simulation architecture that interfaces with the native serving management software (Section 4.1), (2) a pluggable latency estimator (Section 4.2), and (3) an automated configuration tuning pipeline (Section 4.3). During tuning, the optimization algorithm generates candidate configurations, each evaluated by the hybrid simulator through a workload benchmark. Within each simulation, whenever the serving management software issues a model execution call, the estimator returns a predicted latency, allowing the simulation to proceed without actual GPU computation.

4.1 Hybrid Simulation Architecture

The hybrid simulation architecture separates the two major components of an LLM service. The serving management software, including schedulers, request managers, batching logic, and KV-cache management, runs natively on CPU as in a real deployment, while the model inference engine is replaced by a lightweight latency estimator. This boundary follows our observation in Section 3 that serving-management logic dominates fidelity, whereas GPU execution dominates runtime cost.

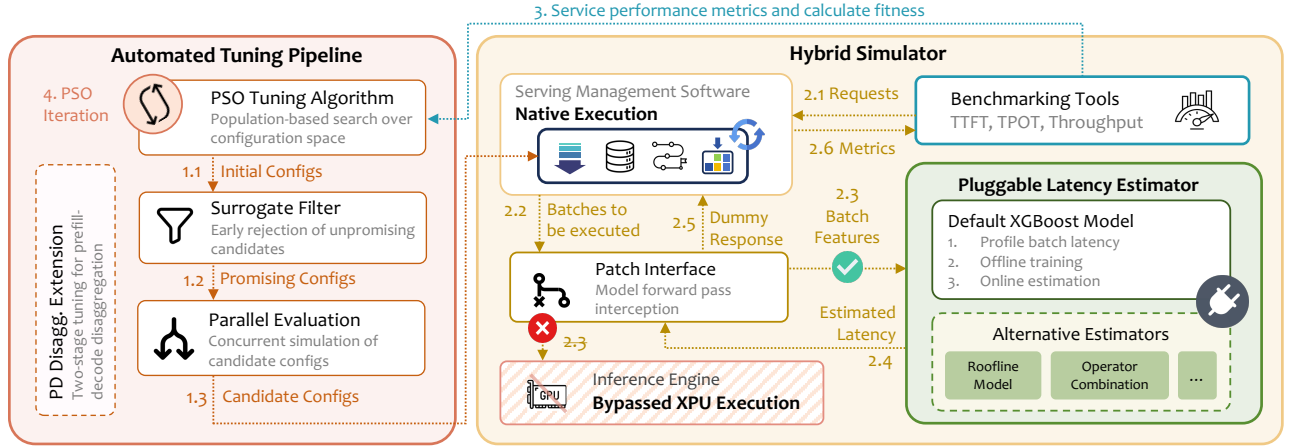


Figure 5: Overview of SIMSERVING Framework

During simulation, the native serving framework first assembles the next batch and invokes the model execution call. Our patch intercepts this call, extracts the current batch state, queries the latency estimator, advances the virtual clock by the predicted duration, and returns control to the native loop without launching the real GPU forward pass. For the scheduling- and memory-management behaviors studied here, returning a placeholder output is sufficient because subsequent control decisions depend on request state, token counts, cache metadata, and simulated timing rather than token semantics.

Algorithm 1: Serving Loop with Patch

```

Input: requests: incoming request stream
1 function ServingLoop(requests):
2   Scheduler.add(requests);
3   while Scheduler.has_pending() do
4     batch ← Scheduler.get_batch();
5     if SIMULATION_MODE then
6       lat ← Estimator.Predict(batch);
7       out ← DummyOutput(batch);
8       AdvanceClock(lat);
9     else
10      out ← ModelExecute(batch);
11    end
12    Scheduler.update(out);
13  end

```

We implement this design through minimal, non-invasive instrumentation at the model execution entry point of the native serving framework. The patch extracts features such as total batch size, per-request prefill and decode token counts, and KV-cache lengths. The same interception point also supports a profiling mode that executes the real GPU path and records latency traces for predictor training. Using one shared patch point keeps feature extraction, profiling, and online simulation aligned with the native framework. Algorithm 1 shows the patched serving loop, where **green lines** denote our instrumentation.

Our open-source implementation currently supports two serving frameworks: vLLM [29] (including vLLM-Ascend [46]) and

MindIE [23]. Adapting a framework requires identifying two components: (1) the model-execution entry point to intercept and (2) batch-feature extraction logic.

The interception point is the function through which the serving framework submits an assembled batch for model execution. It receives the batch representation and returns the model output, while batching, queueing, scheduling, and memory-management decisions are performed outside this function. SIMSERVING therefore replaces only the model-execution call with a latency estimator, leaving the surrounding serving control flow unchanged.

For vLLM, the adaptation required approximately 30 lines of instrumentation around `ModelRunner.execute_model`. Supporting MindIE required a comparable modification at `generate_token`. Although these patches are small, they allow SIMSERVING to reuse approximately 25,000 lines of vLLM’s upstream serving control-plane implementation. Adapting another framework follows the same procedure: locate the batched model-execution entry point, extract the features required by the latency estimator, and replace the underlying device-execution call. The comparable adaptation effort across vLLM and MindIE suggests that the interception design is portable across serving frameworks with a modest engineering cost.

Because the serving-management modules execute natively, SIMSERVING preserves the target framework’s original batching, queueing, scheduling, and memory-management policies. Only the latency of model execution is substituted. Consequently, SIMSERVING retains fidelity to precisely the serving-management behaviors that Study 1 showed are difficult for re-implementation-based simulators to track as the upstream software evolves.

The patch-based design also substantially reduces the effort required to follow version updates and newly introduced serving features. In vLLM’s V1 engine, for example, the wrapped method retains the same signature from v0.10.0 through v0.14.0. Adaptation is required only when an upstream release restructures the intercepted call path. For instance, v0.12.0 introduced a dedicated `_model_forward` helper, requiring the patch to be adjusted to the new call structure. Importantly, such an adjustment updates only the interception mechanism; the associated changes in scheduling

and serving behavior are inherited automatically by executing the updated upstream implementation.

4.2 Pluggable Latency Estimator Interface

The latency-estimator interface abstracts the model inference engine behind a simple contract: it takes the batch-composition features extracted at the patch point and returns a predicted execution latency for the batch. Features include prefill and decode token counts, and KV-cache lengths of all requests in a batch.

This design makes the simulator estimator-agnostic: any backend that maps the native batch state to latency can be plugged in without changing the hybrid simulation architecture. Specifically, a custom predictor implements a single `predict(batch_features)` method and is selected through the `predictor_type` configuration entry. The interface thus separates the stable simulation mechanism from the replaceable prediction mechanism, allowing the same simulator core to support different latency models. For instance, Section 5.3 evaluates a `SimServing-Vidur` variant that reuses Vidur’s predictor under the same interface. Additional estimator choices are discussed in Section 6.1.

In this paper, we instantiate the interface with an XGBoost regressor [10], because it offers a practical balance between prediction accuracy, train cost, inference cost, and engineering simplicity. The predictor is trained offline on profiled execution traces collected from the native framework through the same interception point used during simulation. Section 5 reports the concrete profiling and training setup used in our experiments.

The same interface also makes adaptation explicit. As with existing simulators, moving to a new hardware platform or model architecture still requires new profiling traces, but in our design the simulator core remains unchanged: developers retrain or replace the estimator through the same interface. By contrast, prior simulators (Vidur [1] and APEX [31]) more tightly specify the predictor structure, so substantially different model internals or cluster organizations may also require revising the profiling setup and predictor design. We evaluate the default XGBoost implementation in Section 5.

4.3 Automated Performance Tuning

The hybrid simulator described above provides a fast and accurate way to evaluate serving performance under any given configuration. Building on this capability, we design an automated tuning pipeline that searches the configuration space for settings that maximize throughput while satisfying SLO latency constraints. The pipeline consists of a search algorithm, a fitness function, and an optional acceleration technique.

4.3.1 Search Algorithm. The configuration tuning problem is characterized by a black-box fitness function, a mixed-type search space (continuous and discrete parameters), high dimensionality, and a non-convex landscape with multiple local optima. These characteristics favor population-based, gradient-free optimization methods [6, 21].

We employ Particle Swarm Optimization (PSO) [28] for three reasons: (1) natural parallelism that matches our simulator’s concurrent evaluation capability, (2) parameter-agnostic velocity-based updates that require no domain-specific operator design, and (3)

better scalability than Gaussian Process-based methods [39, 43] in high-dimensional spaces. Each particle maintains its position, velocity, and personal best, with iterative updates balancing exploration and exploitation.

To guide the PSO search, each candidate configuration must be evaluated and assigned a fitness score. We design a fitness function that captures both the primary tuning goal (maximizing throughput) and the constraints (meeting SLO requirements). The fitness function is defined as:

$$\text{fitness} = w_{\text{throughput}} \cdot C_{\text{throughput}} + w_{\text{TTFT}} \cdot C_{\text{TTFT}} + w_{\text{TPOT}} \cdot C_{\text{TPOT}} + w_{\text{success}} \cdot C_{\text{success}} \quad (1)$$

where the weights satisfy $\sum w = 1$. The throughput cost $C_{\text{throughput}}$ is G_{target}/G , where G is the measured throughput and G_{target} is a reference target. The SLO constraint costs the form:

$$C_x = \exp\left(k_x \cdot \left(\frac{p_x}{\text{SLO}_x} - 1\right)\right) \quad (2)$$

where p_x is the measured metric (TTFT, TPOT, or failure rate), SLO_x is the corresponding threshold, and k_x controls the penalty steepness. When $p_x = \text{SLO}_x$, the cost equals 1. Violations ($p_x > \text{SLO}_x$) incur exponentially increasing penalties, while metrics within SLO ($p_x < \text{SLO}_x$) yield costs below 1. This exponential form ensures that SLO violations incur steep penalties, steering the search away from infeasible regions. Configurations meeting SLO constraints but with lower throughput remain viable candidates, as they can be pushed toward higher throughput while staying within bounds.

Notably, the optimizer is itself replaceable: it only proposes configurations and consumes evaluated performance results. Other search methods like Bayesian optimization and genetic algorithms can be plugged in through the same interface. We adopt PSO as the default backend in our implementation, while the tuning pipeline itself can accommodate alternative optimizers.

4.3.2 Surrogate Model. In practice, many candidate configurations are clearly suboptimal, so evaluating all of them with the full hybrid simulator wastes computational budget. We therefore incorporate a surrogate model based on Constrained Bayesian Optimization (CBO) [43] as an early-rejection filter: a Gaussian Process (GP) [39] predicts key performance metrics and estimates whether a candidate is likely to satisfy the SLO constraints before full simulation. During an initial warm-up phase, all candidates are evaluated with the hybrid simulator to collect training data; once sufficient observations are available, the surrogate scores subsequent PSO candidates by predicted feasibility, sending promising ones to the full simulator and assigning clearly unpromising ones estimated fitness values directly. This dual-path evaluation reduces the number of expensive simulation runs while preserving the quality of the final tuning outcome.

4.3.3 Extension to Prefill-Decode Disaggregation. Our framework can also support Prefill-Decode (PD) disaggregation [38, 55], an increasingly important deployment setting in which prefill and decode computations run on separate clusters. This architecture is attractive in practice because the two stages have different compute and communication characteristics, and decoupling them can improve utilization and scaling efficiency. However, it also introduces a new tuning problem: performance now depends not only

on the serving configurations within each cluster, but also on the cross-cluster resource allocation ratio between prefill and decode. Too few prefill nodes increase TTFT by creating a bottleneck in prompt processing, while too few decode nodes degrade TPOT and overall throughput.

We handle this setting with a two-stage extension. We first tune the two clusters independently to obtain strong per-cluster serving configurations and a feasible initial P:D ratio, which reduces the size of the joint search space. We then use this ratio to initialize PSO on the full PD deployment, allowing the optimizer to search nearby allocations under realistic dynamic workloads while still capturing the interaction between the two stages. This decomposition makes the problem tractable without discarding the end-to-end coupling that ultimately determines service quality.

5 Evaluation

In this section, we evaluate SIMSERVING with a focus on the following three research questions.

- **RQ1:** How does SIMSERVING’s hybrid simulation approach compare to prior work in fidelity and efficiency?
- **RQ2:** How effective is SIMSERVING as an end-to-end configuration tuning framework compared to existing methods?
- **RQ3:** How accurate and efficient is the pluggable latency estimator?

We also present ablation studies on the surrogate model and a case study on PD disaggregation to demonstrate the extensibility of our framework.

5.1 Experimental Setup

5.1.1 Hardware and Software Environment. All experiments are conducted on a GPU cluster with 8 NVIDIA H20 GPUs interconnected via NVLink. We use two representative models served on vLLM v0.10.1 as the target serving models: Llama-3.1-8B [19] and Qwen3-30B-A3B [49]. Unless otherwise stated, all experiments target vLLM v0.10.1.

5.1.2 Workloads and Benchmarks. We follow the workload construction methodology from Vidur [1] to ensure our evaluation reflects realistic serving scenarios. The benchmark dataset captures detailed statistics from real-world LLM services, including per-request input and output token lengths and arrival times. The dataset incorporates traces from three diverse sources that represent different batch composition patterns:

- **LMSYS-Chat** [53]: Natural language conversations with balanced input/output lengths, producing *mixed* batches with both prefill and decode operations.
- **ArXiv-Summarization** [13]: Document summarization with long inputs and short outputs, producing *prefill-heavy* batches.
- **BWB** [26]: Translation tasks with long outputs, producing *decode-heavy* batches.

End-to-end benchmarking is conducted by replaying these traces with the framework’s native benchmarking script. All benchmarks are repeated 3 times; we report the mean. Standard deviations are below 2% for throughput and below 5% for latency metrics across all experiments.

Table 1: Tuned configuration knobs, types, ranges, and defaults declared in the search-space configuration. Ratio-typed MindIE knobs are derived from `maxBatchSize`.

Framework	Knob	Type	Range (Default)
vLLM	MAX_NUM_BATCHED_TOKENS	int	8192–65536 (8192)
	MAX_NUM_SEQS	int	32–512 (64)
	CONCURRENCY	int	1–1000 (100)
	REQUESTRATE	float	1–1000 (100)
MindIE	maxBatchSize	int	10–1000
	maxPrefillBatchSize	ratio	0.1–0.7 of batch
	prefillTimeMsPerReq	range	0–1000, step 10
	decodeTimeMsPerReq	range	0–1000, step 10
	supportSelectBatch	bool	{0, 1}
	maxQueueDelayMicrosec.	range	500–10 ⁶ , step 100
	maxPreemptCount	ratio	0–1 of batch
	CONCURRENCY	int	1–1000 (100)
	REQUESTRATE	float	1–1000 (100)

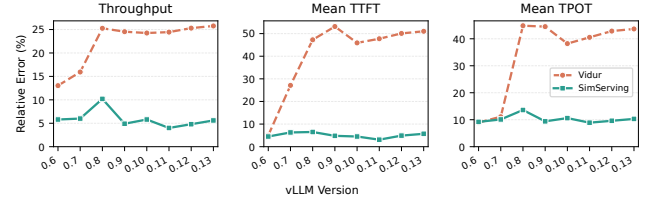


Figure 6: Cross-version simulator error across vLLM versions. Lower error indicates better resilience to software evolution.

5.1.3 Search Space and Tuning Setup. All tuning methods share the search space in Table 1 and the same SLO thresholds. Our PSO tuner uses 10 particles for 5 iterations, with fitness weights 0.4/0.2/0.3/0.1 for throughput/TTFT/TPOT/success. The XGBoost predictor is trained on profiled traces collected from roughly 10K batch samples with a 90:10 train-test split. Other parameters and permethod candidate counts are released with the artifact.

5.1.4 Baseline. Our primary baseline is **Vidur** [1], a state-of-the-art LLM serving simulator. The simulator comprises a component-level latency predictor that estimates GPU execution time by decomposing model architectures into individual operators (**Vidur-Estimator**), and hard-coded serving simulation that re-implements the scheduling, batching, and memory management logic of the target framework. Vidur also provides a tuning tool (**Vidur-Search**) that performs a grid search over the configuration space with binary-search refinement on workload configurations.

5.2 RQ1: Simulation Fidelity and Efficiency

We first evaluate the fidelity and efficiency of our hybrid simulation approach, since tuning quality depends on whether the simulator faithfully reflects the behavior of the target serving software. Here we use SIMSERVING with the default XGBoost estimator and compare it with the Vidur simulator.

5.2.1 Cross-Version Fidelity under Software Evolution. Study 1 established the challenge: re-implementation-based simulation drifts as the serving framework evolves. We therefore examine the same setting from an error-centric perspective. This experiment uses Llama-3.1-8B on the LMSYS-Chat trace and compares prediction error across the vLLM versions shown in Figure 6. Vidur uses the

Table 2: End-to-end metric prediction error (%) of Vidur and our XGBoost simulator compared to native vLLM, evaluated on Llama-3.1-8B and Qwen3-30B-A3B across LMSYS-Chat, ArXiv-Summarization, and BWB. Lower is better.

Workload	Method	Llama-3.1-8B			Qwen3-30B-A3B		
		TPS	TTFT	TPOT	TPS	TTFT	TPOT
LMSYS	Vidur	8.1%	38.0%	15.6%	6.4%	32.2%	74.1%
	SIMSERVING-XGB	5.8%	4.5%	10.5%	1.6%	1.9%	13.6%
Arxiv	Vidur	13.4%	116.2%	40.6%	12.5%	592.3%	54.4%
	SIMSERVING-XGB	1.1%	0.9%	7.8%	0.5%	4.9%	9.5%
BWB	Vidur	18.2%	42.3%	24.5%	24.4%	48.7%	53.1%
	SIMSERVING-XGB	5.7%	4.3%	12.3%	14.5%	9.7%	11.7%

Table 3: End-to-end simulation time (s) and speedup compared to native benchmark for Llama-3.1-8B and Qwen3-30B-A3B across LMSYS-Chat, ArXiv-Summarization, and BWB.

Model	Workload	Native Time	Sim Time	Speedup
Llama-3.1-8B	LMSYS-Chat	533.61	25.19	21.18×
	Arxiv-Summary	1836.48	120	15.30×
	BWB	979.1	283	3.46×
Qwen3-30B-A3B	LMSYS-Chat	257.1	20.98	12.25×
	Arxiv-Summary	1553.4	119	13.05×
	BWB	899.83	231	3.90×

same fixed prediction setup across versions, whereas SIMSERVING follows each target version through native serving-code execution. As shown in Figure 6, Vidur’s error remains sensitive to version changes, while the native-execution design of SIMSERVING removes the main re-implementation drift source by keeping the serving management software aligned with the target framework. We then use the current target version below to validate the end-to-end fidelity achieved by this strategy.

5.2.2 End-to-End Fidelity. For fidelity evaluation, we run each workload benchmark for both models (Llama-3.1-8B and Qwen3-30B-A3B) on three systems: the native vLLM framework (ground truth), the Vidur simulator, and SIMSERVING. We then compare the predicted metrics against the ground truth values: throughput (TPS), mean TTFT, and mean TPOT. Table 2 reports the relative prediction error for each metric.

The results show that SIMSERVING achieves consistently lower prediction errors than Vidur across all metrics, covering 2 models $\times$ 3 workload traces. The improvement is particularly pronounced for latency metrics (TTFT and TPOT), where Vidur’s simplified scheduling logic leads to large mean-latency errors, while SIMSERVING maintains low errors across workloads. This high fidelity is a direct result of our hybrid design, which natively executes the serving management logic of vLLM rather than approximating it with simplified or outdated rules.

5.2.3 Efficiency Evaluation. We measure the wall-clock time required to complete a full throughput benchmark simulation for each model and workload trace. Table 3 shows the simulation time and the speedup factor compared to running the actual benchmark on the native vLLM framework.

SIMSERVING achieves a simulation speedup of up to 21.18 $\times$ compared to running the actual benchmark. This speedup comes from replacing the most time-consuming GPU-bound model forward

Table 4: Wall-clock time (hours) required to complete the tuning process for Llama-3.1-8B and Qwen3-30B-A3B on LMSYS-Chat (reported for Grid-Search, SIMSERVING-Native, and SIMSERVING-XGB).

Method	Llama-3.1-8B	Qwen3-30B-A3B
vLLM-auto_tune (Grid Search)	>24	19.95
SIMSERVING-Native	5.18	3.05
SIMSERVING-XGB	0.39	0.36

pass with a fast XGBoost inference. The lower speedup on the decode-heavy BWB workload is expected: more decodes produce more short iterative steps, and therefore increase the accumulated per-step overhead relative to the GPU time saved in each step.

Summary for RQ1

SIMSERVING achieves consistently better fidelity than Vidur across throughput (TPS) and mean latency metrics (TTFT/TPOT) while providing up to 21.18 $\times$ simulation speedup. The cross-version baseline result further shows why native execution is preferable under software evolution: re-implementation-based simulation drifts as versions change, whereas SIMSERVING removes this source of divergence by keeping the serving management software native.

5.3 RQ2: Configuration Tuning Effectiveness

We next evaluate the simulator’s effectiveness as an end-to-end configuration tuning framework in two aspects: the quality of the final discovered configurations (serving performance) and the efficiency of the tuning process (time and iterations required).

We compare the following methods. **vLLM-auto_tune** is the native tuning utility provided by vLLM. It exhaustively evaluates configurations by running actual benchmarks, serving as the native reference method but incurring substantial time and resource costs. **Vidur-Search** is Vidur’s built-in grid search with binary-search refinement. For SIMSERVING, the framework’s pluggable interface allows the same PSO-based tuning to integrate different backend predictors, yielding three SIMSERVING variants: **SIMSERVING-Vidur** plugs in Vidur’s simulator as the backend predictor, **SIMSERVING-XGB** plugs in our XGBoost latency estimator with native serving execution, and **SIMSERVING-Native** plugs in actual GPU benchmarks, providing an upper bound on tuning quality.

5.3.1 Serving Performance. For RQ2, we focus on a single representative workload trace (LMSYS-Chat) and evaluate two models: Llama-3.1-8B and Qwen3-30B-A3B. The latter is a MoE model and helps validate that our predictor and tuning framework remain effective beyond dense models. After tuning, we deploy the best configuration found by each method on the actual vLLM service and measure its true performance through a full benchmark run. Figure 7 reports performance metrics, and dashed lines indicate the SLO thresholds.

The results show that SIMSERVING-XGB, SIMSERVING-Native, and SIMSERVING-Vidur consistently deliver higher throughput than Vidur-Search and vLLM-auto_tune when evaluated against the same SLO thresholds. SIMSERVING-XGB achieves 6,763.7 TPS on Llama-3.1-8B and 3,324.2 TPS on Qwen3-30B-A3B, improving over

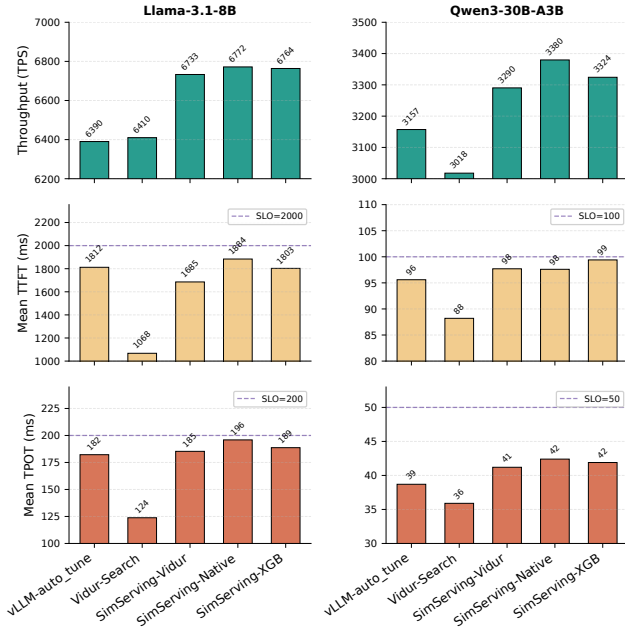


Figure 7: Final serving performance on the LMSYS-Chat workload trace for two models. Higher throughput is better, and lower latency is better.

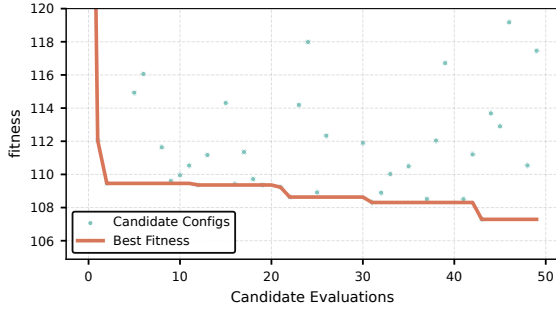


Figure 8: Convergence curve of SimServing-XGB on the LMSYS-Chat trace for Llama-3.1-8B.

Vidur-Search (6,410.0 and 3,017.8 TPS) by 5.5% and 10.2%, respectively, while staying within the SLO constraints. SIMSERVING-Native reaches the highest SLO-satisfying throughput on both models (6,771.7 and 3,379.6 TPS), serving as an upper bound, and the SIMSERVING-Vidur variant (6,733.0 and 3,290.3 TPS) still outperforms Vidur-Search, indicating that both the search algorithm and the simulation fidelity contribute to tuning effectiveness. Notably, all three SIMSERVING variants achieve similar throughput despite using different backend predictors, indicating that our PSO-based tuning framework works consistently across the evaluated backends.

5.3.2 Tuning Efficiency. Beyond the quality of the final configuration, we also evaluate how efficiently each method reaches a good solution. Figure 8 shows the convergence curve of SIMSERVING-XGB on the LMSYS-Chat trace for Llama-3.1-8B, plotting the best configuration quality (measured by simulated fitness) against the number of evaluated candidates.

SIMSERVING-XGB converges to a near-optimal configuration within approximately 50 candidate evaluations (5 PSO iterations with 10 particles each), while grid search would require 5120 evaluations to exhaustively explore the same configuration space. This faster convergence highlights the sample efficiency of our PSO-based search for exploring the configuration space and quickly identifying high-quality candidates.

Table 4 summarizes the total wall-clock time required to complete end-to-end tuning. In our setting, SIMSERVING-XGB is significantly faster than SIMSERVING-Native and exhaustive grid search, demonstrating the efficiency benefit of simulation-based evaluation. We therefore only report wall-clock time comparisons against native grid search (vLLM-auto_tune), SIMSERVING-Native, and SIMSERVING-XGB. For a fresh deployment of SIMSERVING-XGB, the one-time setup adds 36 minutes of profiling and about 10 s of XGBoost training, bringing the end-to-end cost on Llama-3.1-8B to about 59.5 minutes (with 0.39 h of tuning), still far below exhaustive native tuning.

Summary for RQ2

SIMSERVING finds configurations that achieve up to 10.2% higher throughput than Vidur-Search, converging within 50 candidate evaluations. The wall-clock tuning time is substantially lower than both SIMSERVING-Native and exhaustive grid search. Both the hybrid simulation architecture and the PSO-based search algorithm contribute to this effectiveness.

5.4 RQ3: Latency Estimator Evaluation

We evaluate the accuracy and overhead of the latency estimator, which is a pluggable component that enables fast simulation while maintaining fidelity. As discussed in Section 4, our framework supports multiple estimation methods through a unified interface. We compare two pluggable estimator implementations: our XGBoost-based predictor and Vidur’s component-level estimator, to assess their prediction quality and computational cost.

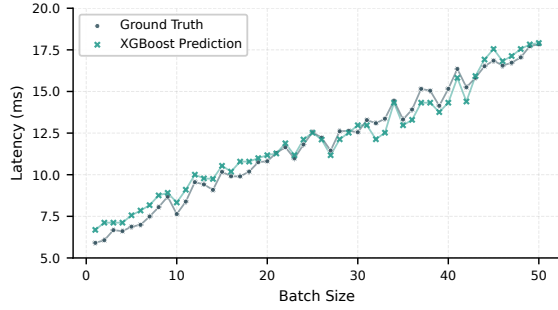
5.4.1 Prediction Accuracy. To isolate the accuracy of predictor from serving-layer behavior, we collect a held-out test set of batch execution traces from actual vLLM runs under the same hardware and target-version setting. Each trace contains batch-composition features (prefill tokens, decode tokens, and KV-cache lengths) and the measured GPU execution time. We compare XGBoost and Vidur predictions against these ground-truth measurements.

Table 5 reports mean absolute percentage error (MAPE) for each predictor across the three workloads and two models: LMSYS-Chat produces mixed batches with balanced prefill and decode operations, ArXiv-Summarization is prefill-heavy due to long input documents with short summaries, and BWB is decode-heavy with short prompts and long outputs.

The XGBoost predictor achieves a MAPE range of 3.73%–10.22% across all workloads and batch types, with a 6.25% average. Vidur’s component-level estimator has higher errors, especially on mixed batches containing both prefill and decode requests, because it cannot capture the interactions that arise when different request types share GPU resources within the same batch. XGBoost learns these end-to-end interactions directly from training data.

Table 5: Batch-level inference latency prediction error (MAPE) for Llama-3.1-8B and Qwen3-30B-A3B across LMSYS-Chat, ArXiv-Summarization, and BWB. Lower is better.

Workload	Llama-3.1-8B		Qwen3-30B-A3B	
	Vidur	XGB	Vidur	XGB
LMSYS-Chat (Mixed)	12.96%	4.81%	39.96%	7.65%
Arxiv-Summary (Prefill)	14.80%	6.30%	37.25%	10.22%
BWB (Decode)	9.12%	3.73%	19.84%	4.78%
Overall	12.29%	4.95%	32.35%	7.55%

**Figure 9: Measured and XGBoost-predicted GPU execution time across sampled batches of different batch sizes.****Table 6: Estimator error under out-of-distribution stress tests (MindIE on Ascend 910B, Llama-3.1-8B). The in-distribution baseline on this platform is about 3% MAPE.**

OOD Axis	Condition	#Batches	MAPE
Batch-composition shift	output len. between profiled 128/512	8,219	3.06%
Decode-length extrapolation	1,024-token decode (2× max)	8,708	2.94%
Arrival-rate shift	concurrency 32 (2–4× training)	3,225	8.37%

Table 7: Ablation study: Effect of surrogate model on tuning efficiency for Llama-3.1-8B on LMSYS-Chat-1m.

Method	Sims	Time (s)	TPS	Speedup
SIMSERVING-XGB (w/ surrogate)	23	2341.0	6763.7	2.67×
SIMSERVING-XGB (w/o surrogate)	72	6238.1	6724.9	1.00×

Figure 9 plots measured GPU execution time and XGBoost predictions for sampled batches across batch sizes. The predicted curve closely follows the ground-truth trend, providing an intuitive view of predictor accuracy.

5.4.2 Prediction Overhead. We measure the time required for a single latency prediction call. The XGBoost model takes an average of 1.92 ms per prediction on CPU. Given that a typical serving simulation involves thousands of batch execution steps, this overhead is still small compared to the millisecond-scale latencies being predicted.

5.4.3 Robustness. To test the estimator beyond the profiled distribution, we evaluate on Ascend 910B with Llama-3.1-8B, training on 108,591 batch samples merged from three workloads and scoring on unseen workloads that shift with three different patterns: batch composition, decode length, and arrival rate. As shown in

Table 8: Case study: Configuration tuning for PD disaggregation on Llama-3.1-8B with LMSYS-Chat-1m on 8 nodes.

Method	Manual (1:1)	Manual (1:3)	SIMSERVING-Auto
P:D Ratio.	4P:4D	2P:6D	3P:5D
Throughput (tokens/s)	23392.6	25529.9	27375.9

Table 6, prediction error under all three shifts remains close to the in-distribution baseline on this platform.

Summary for RQ3

The XGBoost predictor, as the default implementation of our pluggable estimator interface, achieves 3.73%–10.22% MAPE on batch latency prediction (6.25% average), outperforming Vidur’s component-level estimator. The prediction overhead is only 1.92 ms per call, and prediction error stays close to the in-distribution level under out-of-distribution stress tests.

5.5 Ablation Study: Surrogate Model

To evaluate the contribution of the surrogate model (Section 4.3.2) in accelerating the tuning process, we compare the tuning efficiency with and without the early-rejection filter. Unless otherwise noted, this ablation uses the same setup as RQ2.

The surrogate model reduces the number of expensive full simulations by filtering out clearly infeasible configurations early. This results in faster convergence and significant speedup in the tuning process without sacrificing the quality of the final configuration. As shown in Table 7, the surrogate-enabled variant achieves comparable final TPS while requiring fewer full simulations and less wall-clock time.

5.6 Extension Case Study: PD Disaggregation

To demonstrate extensibility beyond the default serving setup, we apply SIMSERVING to Prefill-Decode (PD) disaggregation, where prefill and decode stages run on separate GPU clusters. We use the same hardware and tuning setup as RQ2, and then adapt the simulator and search space to the disaggregated deployment.

Table 8 shows results for Llama-3.1-8B on 8 nodes. SIMSERVING-Auto discovers a 3P:5D configuration achieving 27,375.9 TPS, outperforming manual 1:1 ratio (4P:4D, 23,392.6 TPS) by 17.0% and 1:3 ratio (2P:6D, 25,529.9 TPS) by 7.2%. This result shows that the same simulation-and-tuning interface extends to disaggregated deployments.

5.7 Industrial Practice on MindIE

SIMSERVING’s MindIE support has been fully implemented and validated in industrial practice. On an Ascend 910B NPU, the simulated execution time for MindIE 2.0.RC1 serving Llama-3.1-8B on LMSYS-Chat tracks the measured time within 1.2% relative error (27.25 s versus 26.94 s over 200 requests), in line with the fidelity on vLLM. In an industrial deployment on 910B NPUs, SIMSERVING tuned Qwen3-32B under an SLO of TTFT < 3 s and TPOT < 0.05 s, improving throughput from 720 to 750 tokens/s (+4.2% over the engineer-tuned configuration). For reproducibility, the artifact includes the MindIE plugin-manager patch, simulation plugin, and tuned configuration space.

6 Discussion

6.1 Alternative Latency Estimation Approaches

Our pluggable estimator interface allows alternative latency estimation methods to be substituted without modifying the simulator core. Roofline-based analytical models [47] require no profiling but oversimplify GPU behaviors such as kernel fusion. Operator-level composition [1] offers interpretability but suffers from error accumulation and manual rule maintenance. Neural network predictors could capture more complex patterns but need larger training datasets and may be less stable. Our evaluation (Section 5.4) shows that XGBoost [10] provides a practical sweet spot: 6.25% average MAPE with modest profiling requirements (about 36 minutes).

6.2 Threats to Validity

Internal Validity. Our hybrid simulation assumes that the serving management software’s control flow does not depend on actual generated token values and semantics. This assumption holds for standard continuous batching and chunked prefill, but would be violated by techniques such as grammar-guided decoding, where generation length and control flow are coupled to output content. Additionally, the XGBoost predictor is trained from profiled traces collected for the target deployment setting. While Section 5.4 reports only modest error increases under controlled distribution shifts, we do not claim accuracy for batch compositions far outside the profiled operating range. In such cases, additional profiling and retraining may be required.

External Validity. Our framework supports both vLLM and MindIE (Ascend NPUs only), but the full controlled tuning comparison is conducted only on vLLM due to NPU hardware availability. Section 5.7 reports an industrial MindIE deployment, but they remain outside the controlled cross-framework tuning comparison, which we leave to future work. Additionally, adapting to other frameworks (e.g., SGLang) requires relocating the instrumentation point and depends on the framework exposing a clear model-execution boundary from which batch features can be extracted. The XGBoost predictor requires only re-profiling for new architectures without code changes.

6.3 Future Work

We see three directions for future work. First, the current PSO-based search could be replaced by LLM-guided agentic tuning workflows, where a language model acts as both candidate generator and evaluation interpreter, potentially reducing the number of simulation trials needed. Second, our hybrid simulation design could be applied beyond LLM serving to other ML inference systems (e.g., diffusion model serving, multi-modal pipelines) that face similar tensions between evolving software stacks and expensive hardware evaluation. Third, validating SIMSERVING’s tuning effectiveness across additional serving frameworks such as SGLang would strengthen the generality claims of the native-execution approach.

7 Related Work

LLM Inference Simulation. Vidur [1], APEX [31], and Frontier [15] model serving performance through operator-level profiling and scheduling simulation, with Frontier further targeting distributed and MoE settings. Fast End-to-End Simulation [33],

LLMServingSim [12], and LLMServingSim2.0 [11] instead emphasize hardware-software co-simulation, focusing respectively on iteration-level serving, heterogeneous hardware integration, and accelerated full-stack simulation; related extensions study uncertainty quantification [40] and workload characterization [30]. Despite these differences, they all re-implement or simplify the serving management software, creating a static semantic copy that cannot track rapid upstream evolution and therefore suffers from the fidelity drift shown in Section 3. Our work addresses this limitation by executing the native serving management software directly.

Most closely related, Revati [2] likewise favors native execution over re-implementation for fidelity, but the two systems target different layers. Revati provides a general GPU-free emulation substrate through CUDA-level interception and distributed time synchronization, whereas SIMSERVING targets configuration tuning under rapidly evolving serving software through framework-level interception at the model-execution entry point and a pluggable latency-estimator interface that integrates naturally with automated tuning.

Automated Configuration Tuning. Automated configuration tuning has been widely studied for databases [44], cloud systems [57], and big-data analytics [22]. In software engineering area, learning-based methods predict configurable-system performance [17, 20, 24, 41, 45], including workload-aware variants [35], while search strategies include Bayesian optimization [6], multi-objective optimization [8], causal-inference-guided tuning [7], surrogate-model analysis [9], and configuration-space reduction [5, 16, 27, 32, 36]. For LLM serving, AIConfigurator [48] uses theoretical performance modeling, Pimp My LLM [58] applies software-variability modeling to inference hyperparameters, and Vidur [1] couples its simulator with grid search. These approaches largely assume a stable evaluation oracle; in rapidly evolving LLM serving software, re-implementation-based evaluators can drift and silently invalidate tuning results. Our work complements existing tuning strategies by providing an evolution-resilient oracle through native execution of the serving management software.

8 Conclusion

We identified the re-implementation pitfall in existing LLM serving simulators and proposed SIMSERVING, a hybrid simulation framework that natively executes the serving management software while replacing GPU computation with a pluggable latency estimator. Evaluation shows that this design substantially improves simulation fidelity, enables effective configuration tuning, and automatically stays synchronized with upstream framework evolution without manual maintenance.

Acknowledgments

We thank the MSModeling team for their valuable discussions and support. In particular, we thank Dengkai Wang, Jintao Xu, Kai Fang, and Han Yao for their help in developing the prototype system and conducting extensive testing in practical deployment settings.

The work described in this paper was supported by two grants from the Research Grants Council of the Hong Kong Special Administrative Region, China: (1) No. SRFS2425-4S03 of the Senior Research Fellow Scheme, and (2) No. CUHK 14209124 of the General Research Fund.

Data Availability Statement

The artifact (<https://doi.org/10.6084/m9.figshare.31899004>) contains the SIMSERVING source code, experiment scripts, plotting scripts, and the data needed to reproduce the reported results, including the instrumentation patches for vLLM and MindIE.

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. *Proceedings of Machine Learning and Systems* 6 (2024), 351–366.
- [2] Amey Agrawal, Mayank Yadav, Sukrit Kumar, Anirudha Agrawal, Garv Ghai, Souradeep Bera, Elton Pinto, Sirish Gambhira, Mohammad Adain, Kasra Sohrab, Chus Antonanzas, and Alexey Tumanov. 2026. Revati: Transparent GPU-Free Time-Warp Emulation for LLM Serving. *arXiv:2601.00397* [cs] doi:10.48550/arXiv.2601.00397
- [3] Anthropic. 2025. The Claude 4 Model Family: Opus, Sonnet. <https://www.anthropic.com/news/claude-4>. Accessed: 2026-03-01.
- [4] Anthropic. 2025. Claude Code. <https://docs.anthropic.com/en/docs/claude-code>. Accessed: 2026-03-01.
- [5] Liang Bao, Xin Liu, Ziheng Xu, and Baoyin Fang. 2018. AutoConfig: Automatic Configuration Tuning for Distributed Message Systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 29–40. doi:10.1145/3238147.3238175
- [6] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. IEEE Press, Madrid, Spain, 1198–1209. doi:10.1109/ICSE43902.2021.00110
- [7] Pengzhou Chen and Tao Chen. 2025. PromiseTune: Unveiling Causally Promising and Explainable Configuration Tuning. *arXiv:2507.05995* [cs] doi:10.1145/3744916.3764552
- [8] Pengzhou Chen, Tao Chen, and Miqing Li. 2024. MMO: Meta Multi-Objectivization for Software Configuration Tuning. *IEEE Transactions on Software Engineering* 50, 6 (June 2024), 1478–1504. doi:10.1109/TSE.2024.3388910
- [9] Pengzhou Chen, Jingzhi Gong, and Tao Chen. 2025. Accuracy Can Lie: On the Impact of Surrogate Model in Configuration Tuning. *IEEE Transactions on Software Engineering* 51, 2 (Feb. 2025), 548–580. doi:10.1109/TSE.2025.3525955
- [10] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 785–794.
- [11] 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* (2025), 1–4. *arXiv:2511.07229* [cs] doi:10.1109/LCA.2025.3628325
- [12] 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 *2024 IEEE International Symposium on Workload Characterization (IISWC)*, 15–29. *arXiv:2408.05499* [cs] doi:10.1109/IISWC63097.2024.00012
- [13] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. 2018. A Discourse-Aware Attention Model for Abstractive Summarization of Long Documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, Marilyn Walker, Heng Ji, and Amanda Stent (Eds.). Association for Computational Linguistics, New Orleans, Louisiana, 615–621. doi:10.18653/v1/N18-2097
- [14] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008
- [15] Yicheng Feng, Xin Tan, Kin Hang Sew, Yimin Jiang, Yibo Zhu, and Hong Xu. 2025. Frontier: Simulating the Next Generation of LLM Inference Systems. In *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems*. ACM, Seoul Republic of Korea, 25–30. doi:10.1145/3766882.3767173
- [16] Yanjie Gao, Yonghao Zhu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2021. Resource-Guided Configuration Space Reduction for Deep Learning Models. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. IEEE Press, Madrid, Spain, 175–187. doi:10.1109/ICSE43902.2021.00028
- [17] Jingzhi Gong and Tao Chen. 2024. Deep configuration performance learning: A systematic survey and taxonomy. *ACM Transactions on Software Engineering and Methodology* 34, 1 (2024), 1–62.
- [18] Google DeepMind. 2025. Gemini 3 Pro Model Card. <https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf>. Accessed: 2026-03-01.
- [19] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The Llama 3 Herd of Models. *arXiv:2407.21783* [cs.AI] <https://arxiv.org/abs/2407.21783>
- [20] Huang Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 1095–1106. doi:10.1109/ICSE.2019.00113
- [21] Haochen He, Zhouyang Jia, Shanshan Li, Yue Yu, Chenglong Zhou, Qing Liao, Ji Wang, and Xiangke Liao. 2022. Multi-intention-aware configuration selection for performance tuning. In *Proceedings of the 44th International Conference on Software Engineering*, 1431–1442.
- [22] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A self-tuning system for big data analytics. In *Cidr*, Vol. 11. 261–272.
- [23] Huawei Technologies. 2024. MindIE: Mind Inference Engine for Large Language Model Serving on Ascend NPU. MindSpore Transformers Documentation. <https://www.mindspore.cn/mindformers/docs/en/master/guide/deployment.html>
- [24] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to Sample: Exploiting Similarities across Environments to Learn Performance Models for Configurable Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 71–82. doi:10.1145/3236024.3236074
- [25] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A Survey on Large Language Models for Code Generation. *ACM Transactions on Software Engineering and Methodology* 35, 2 (Feb. 2026), 1–72. doi:10.1145/3747588
- [26] Yuchen Eleanor Jiang, Tianyu Liu, Shuming Ma, Dongdong Zhang, Mrinmaya Sachan, and Ryan Cotterell. 2023. Discourse Centric Evaluation of Machine Translation with a Densely Annotated Parallel Corpus. (2023). doi:10.48550/ARXIV.2305.11142
- [27] Zhuangwei Kang, Yogesh D. Barve, Shunxing Bao, Abhishek Dubey, and Anirudha Gokhale. 2023. DMSConfig: Automated Configuration Tuning for Distributed IoT Message Systems Using Deep Reinforcement Learning. *arXiv:2302.09146* [cs] doi:10.48550/arXiv.2302.09146
- [28] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization (PSO). In *Proc. IEEE international conference on neural networks, Perth, Australia*, Vol. 4. 1942–1948.
- [29] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. ACM, Koblenz Germany, 611–626. doi:10.1145/3600006.3613165
- [30] Malgorzata Lazuka, Andreea Anghel, and Thomas Parnell. 2024. Llm-pilot: Characterize and optimize performance of your llm inference services. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–18.
- [31] Yi-Chien Lin, Woosuk Kwon, Ronald Pineda, and Fanny Nina Paravecino. 2024. Apex: An extensible and dynamism-aware simulator for automated parallel execution in llm serving. *arXiv preprint arXiv:2411.17651* (2024).
- [32] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shivnath Babu. 2019. Speedup your analytics: Automatic parameter tuning for databases and big data systems. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1970–1973.
- [33] Jiacheng Ma, Jonas Kaufmann, Emilien Guandalino, Rishabh Iyer, Thomas Bourgeat, and George Candea. 2025. Fast End-to-End Performance Simulation of Accelerated Hardware-Software Stacks. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. ACM, Lotte Hotel World Seoul Republic of Korea, 341–358. doi:10.1145/3731569.3764825
- [34] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2025. Large Language Models: A Survey. *arXiv:2402.06196* [cs] doi:10.48550/arXiv.2402.06196
- [35] Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Sven Apel, and Norbert Siegmund. 2023. Analysing the Impact of Workloads on Modeling the Performance of Configurable Software Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2085–2097. doi:10.1109/ICSE48619.2023.00176
- [36] Jeho Oh, Don Batory, and Rubén Heradio. 2024. Finding Near-optimal Configurations in Colossal Spaces with Statistical Guarantees. *ACM Transactions on Software Engineering and Methodology* 33, 1 (Jan. 2024), 1–36. doi:10.1145/3611663
- [37] OpenAI, Josh Achiam, Steven Adler, et al. 2024. *GPT-4 Technical Report*. Technical Report *arXiv:2303.08774*. *arXiv:2303.08774* [cs] doi:10.48550/arXiv.2303.08774
- [38] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative Llm Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.
- [39] Carl Edward Rasmussen. 2003. Gaussian processes in machine learning. In *Summer school on machine learning*. Springer, 63–71.
- [40] Kaustabha Ray, Nelson Mimura Gonzalez, Bruno Wassermann, Rachel Tzoref-Brill, and Dean H Lorenz. 2025. Statistical Modeling and Uncertainty Estimation of LLM Inference Systems. *arXiv preprint arXiv:2505.09319* (2025).

- [41] Atrisha Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 342–352. doi:10.1109/ASE.2015.45
- [42] Aaditya Singh, Adam Fry, Adam Perelman, et al. 2025. OpenAI GPT-5 System Card. arXiv:2601.03267 [cs] doi:10.48550/arXiv.2601.03267
- [43] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems* 25 (2012).
- [44] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. doi:10.1145/3035918.3064029
- [45] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. arXiv:2101.05362 [cs] doi:10.48550/arXiv.2101.05362
- [46] vLLM Project. 2025. vLLM Ascend: Community-Maintained Hardware Plugin for vLLM on Ascend NPU. GitHub repository. <https://github.com/vllm-project/vllm-ascend>
- [47] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785
- [48] Tianhao Xu, Yiming Liu, Xianglong Lu, Yijia Zhao, Xuting Zhou, Aichen Feng, Yiyi Chen, Yi Shen, Qin Zhou, Xumeng Chen, et al. 2026. AIConfigurator: Lightning-fast configuration optimization for multi-framework LLM serving. *arXiv preprint arXiv:2601.06288* (2026).
- [49] An Yang, Anfeng Li, Baosong Yang, et al. 2025. *Qwen3 Technical Report*. Technical Report arXiv:2505.09388. arXiv:2505.09388 [cs] doi:10.48550/arXiv.2505.09388
- [50] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.
- [51] Wayne Xin Zhao, Kun Zhou, Junyi Li, et al. 2026. A Survey of Large Language Models. arXiv:2303.18223 [cs] doi:10.48550/arXiv.2303.18223
- [52] Ranran Zhen, Juntao Li, Yixin Ji, Zhenlin Yang, Tong Liu, Qingrong Xia, Xinyu Duan, Zhefeng Wang, Baoxing Huai, and Min Zhang. 2025. Taming the Titans: A Survey of Efficient LLM Inference Serving. In *Proceedings of the 18th International Natural Language Generation Conference*, Lucie Flek, Shashi Narayan, Lê H`ong Phng, and Jiahuan Pei (Eds.). Association for Computational Linguistics, Hanoi, Vietnam, 522–541.
- [53] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. arXiv:2309.11998 [cs] doi:10.48550/arXiv.2309.11998
- [54] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H. Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, and Joseph E. Gonzalez. 2024. Sglang: Efficient Execution of Structured Language Model Programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.
- [55] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe : Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.
- [56] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, Shengen Yan, Guohao Dai, Xiaoping Zhang, Yuhan Dong, and Yu Wang. 2024. A Survey on Efficient Inference for Large Language Models. arXiv:2404.14294 [cs] doi:10.48550/arXiv.2404.14294
- [57] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 338–350. doi:10.1145/3127479.3128605
- [58] Nada Zine, Clément Quinton, and Romain Rouvoy. 2026. Pimp My LLM: Leveraging Variability Modeling to Tune Inference Hyperparameters. *arXiv preprint arXiv:2602.17697* (2026).

Received 2026-03-27; accepted 2026-06-18