AGENTSIMARENA: BENCHMARKING LLM SERVING SIMULATORS FOR AGENT WORKLOADS

Anonymous authors
Paper under double-blind review

ABSTRACT

Serving simulators can reduce the cost of exploring LLM-agent deployments, but inaccurate predictions can lead to poor deployment choices. Evaluating these simulators is difficult because each predicted call completion determines when subsequent work arrives, changing future contention. We introduce AgentSimArena, a benchmark that preserves this feedback while comparing simulator predictions with real executions under matched workload and deployment constraints. Across six simulators, eight deployments, and five arrival rates, we evaluate 240 predictions against 40 vLLM reference runs. LLMServingSim 2.0 achieves the lowest mean absolute relative error in mean program completion time, at 7.1%; DynoSim achieves 10.6%, with a median simulation runtime of 6.86 s versus 1,224.26 s for LLMServingSim. No simulator remains within 10% error across all settings. In one observed deployment comparison, following a simulator's recommendation selects an alternative with $6.07\times$ the measured mean program completion time of the other deployment. Arena documents integration extensions, modeling differences, and available preparation-cost evidence, providing a basis for workload-specific simulator selection.

1 INTRODUCTION

Large language model (LLM) agents alternate model calls with tool execution (Yao et al., 2023). Each completion determines when subsequent work becomes ready, which requests it competes with, and whether context can be reused (Abhyankar et al., 2024; Luo et al., 2025). Evaluating deployments therefore requires measuring complete programs across models, hardware, and loads—an expensive process on real GPUs. Serving simulation offers a practical alternative if its predictions capture these interactions.

An incorrect prediction can lead to a costly choice. In one observed comparison using two B200 GPUs, SplitwiseSim recommends two independent replicas over one tensor-parallel instance for Llama-3.1-70B at 0.10 programs/s. Yet its preferred deployment has $6.07\times$ the measured mean program completion time (Appendix G). The alternatives differ in cache capacity and placement as well as parallelism, so this is not an isolated parallelism comparison. This motivates our question: *Before using a simulator to guide agent deployment, how can we determine whether its predictions are accurate enough for the intended workload?*

Existing serving simulators model inference execution, batching, and contention at different levels of detail (Agrawal et al., 2024a; Cho et al., 2024; Lin et al., 2025; Ding et al., 2026). Their evaluations also differ in workloads, hardware, reference engines, and metrics. Published errors under these protocols provide limited guidance for the deployment decision above. We introduce AgentSimArena (Arena), a common protocol that preserves completion-dependent agent execution, documents simulator adaptations, and compares program-level prediction accuracy and execution cost against real serving measurements. Building this comparison requires addressing three difficulties.

Prediction changes future demand. If a simulator predicts that a call finishes early, the next call enters the queue early and changes contention for other programs. Replaying calls at timestamps recorded from real execution suppresses this feedback. Arena instead releases each successor after its predecessor completes and its tool delay elapses, independently in simulation and real serving. It uses native dependency support (Ding et al., 2026; Cho et al., 2026) where available and integration extensions elsewhere.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Matching inputs does not automatically match the experiment. The same model and trace can execute under different memory budgets, batching limits, and replica configurations. Arena uses shared deployment constraints and measured reference capacity to align the experiment. It records differences in cache behavior, scheduling, and routing, distinguishing native mechanisms from integration adaptations.

Accuracy alone does not determine usefulness. A simulator's value depends on how long a prediction takes and what preparation it requires. Arena reports program-level error, execution time, and available preparation costs separately. Separating reusable profiling from repeated simulation exposes the costs relevant to both one deployment check and a larger sweep.

We evaluate six simulators across eight deployments, two GPU platforms, three models, and three workloads. Five arrival rates yield 240 predictions against 40 vLLM executions. LLMServingSim achieves 7.1% mean absolute relative error in mean program completion time; DynoSim achieves 10.6% with a median runtime approximately $178\times$ shorter. No simulator stays within 10% error throughout the matrix.

Our contributions are:

- **A common execution and measurement protocol.** Arena preserves completion-dependent call release independently in simulation and real serving, comparing complete programs under matched constraints.
- **Aligned deployments and documented adaptations.** Measured resource limits and a capability comparison expose differences in cache behavior, scheduling, and simulator integration.
- **A matched accuracy–cost study.** Across 240 predictions, DynoSim is nondominated in 39 of 40 settings, while LLMServingSim has the lowest aggregate error. Profiling analysis and a one-anchor calibration diagnostic identify directions for improving fidelity.

2 Related Work

Simulation for serving-system exploration. Serving simulators make deployment choices tractable by estimating performance before committing GPU resources. Splitwise uses simulation to explore prefill/decode separation and releases its infrastructure as SplitwiseSim (Patel et al., 2023); APEX explores dynamic batching and parallel execution plans (Lin et al., 2025); and TokenSim supports hardware/software exploration through scheduling and memory models (Wu et al., 2025). These tools also differ in how they obtain execution times. Vidur combines operator profiling with predictive models (Agrawal et al., 2024a), LLMServingSim couples serving dynamics with hardware simulation (Cho et al., 2024), and LLM-Emu replaces GPU execution within vLLM with profile-derived wall-clock delays (Da & Kalyvianaki, 2026). Frontier and LLMServingSim 2.0 extend the deployment space to co-located, heterogeneous, or disaggregated infrastructure (Feng et al., 2026; Cho et al., 2026). Arena evaluates the predictive evidence behind such exploration: how closely each simulator reproduces program completion time for a specified deployment, and what preparation and execution costs accompany that accuracy.

Agent execution and serving mechanisms. Agent programs connect serving decisions across calls through dependencies, tool waits, and reusable context. SGLang exploits shared prefixes across language-model calls (Zheng et al., 2024), InferCept manages interruptions for external interactions (Abhyankar et al., 2024), and Autellix uses program-level information for scheduling and placement (Luo et al., 2025). Their performance also depends on request-level mechanisms: iteration-level scheduling in Orca (Yu et al., 2022), KV-cache allocation and sharing in vLLM (Kwon et al., 2023), chunked prefill in Sarathi-Serve (Agrawal et al., 2024b), and prefill/decode disaggregation in DistServe (Zhong et al., 2024). Dependency-aware execution is already supported by simulators such as DynoSim, whose sessions include completion-dependent requests and inter-turn delays (Ding et al., 2026). Arena builds on these mechanisms to enforce a common completion-dependent release rule in simulation and real serving, while documenting cache, scheduling, and integration differences. SWE-bench and BFCL supply software-engineering and function-calling task families (Jimenez et al., 2024; Patil et al., 2025); Arena uses derived fixed traces to study their serving performance.

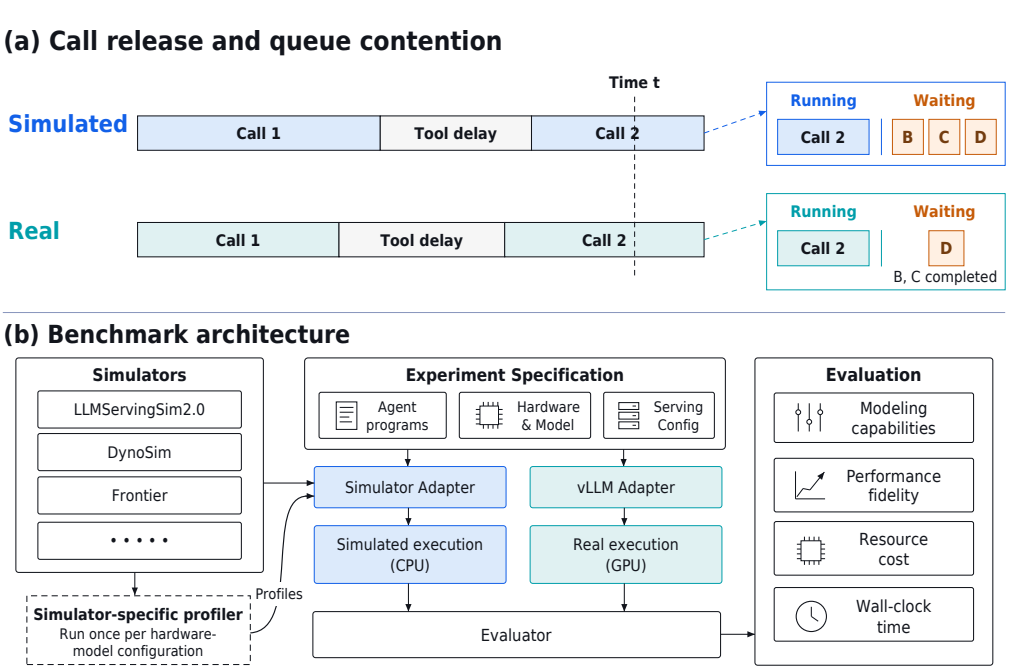


Figure 1: Matched inputs and independent execution: each branch releases dependent calls using its own completion times.

Validation and comparable benchmark evidence. Validation infrastructure makes simulation predictions testable against measured execution. Accel-Sim uses microbenchmarks and hardware-counter comparisons for GPU-model validation (Khairy et al., 2020), while Vidur-Bench provides inference workloads, scheduling policies, and performance metrics for serving evaluation (Agrawal et al., 2024a). Workload construction is central to this comparison: open and closed arrival models can yield substantially different performance at comparable load (Schroeder et al., 2018). Arena combines externally scheduled program arrivals with completion-dependent calls, allowing each execution's completion times to determine its subsequent demand. Its contribution is a common cross-simulator protocol that aligns deployment constraints, discloses adaptations and profiling inputs, and compares complete-program predictions with real measurements. Reporting accuracy, runtime, and available preparation-cost evidence separately makes the comparison useful for selecting a simulator for an intended workload and deployment.

3 AgentSimArena

3.1 Overview and Goals

Arena evaluates whether a simulator can reproduce complete-program latency under a specified deployment. Its central methodological choice is to match the conditions of an experiment without fixing its outcome. Both branches receive the same prescribed work, deployment constraints, and call-release rule; each determines its own completion times and subsequent demand. Figure 1 shows this separation. Measured engine capacity is an input constraint for deployment alignment, while measured completion times are evaluation targets. The simulator predicts execution using its own timing model and profiles.

3.2 Matched Workloads and Deployments

The workload separates externally specified demand from execution-dependent demand. Arena fixes initial program arrivals, call inputs, requested output lengths, and intervening tool delays. Later calls become ready as their predecessors complete. This holds the prescribed work constant while allowing each serving system to change when that work enters contention.

Deployment alignment requires more than a model and GPU count. Tensor parallelism distributes one instance across devices, whereas replication creates independent instances with separate serving state. Arena specifies topology and serving limits, using measured per-instance KV capacity. Capacity determines how much context can remain resident (Kwon et al., 2023); allocation, eviction, and reuse determine how that capacity affects execution. Arena aligns the resource constraint while retaining each simulator's cache and scheduling behavior. Appendix H gives an example specification.

3.3 Dependency-Preserving Replay

Let a_p be program p's arrival, $d_{p,k}$ the tool delay after call k, and $c_{p,k}^{(b)}$ its completion in system b. Arena releases calls according to

$$r_{p,1}^{(b)} = a_p, \qquad r_{p,k+1}^{(b)} = c_{p,k}^{(b)} + d_{p,k}. \tag{1}$$

The same rule applies independently to simulation and real serving. Programs arrive externally, while calls within a program form a completion-dependent chain; arrival-model choices affect queueing behavior (Schroeder et al., 2018).

Suppose a simulator finishes a call earlier than real serving. Its successor enters the queue earlier, potentially joining a different batch and changing contention for other programs. Releasing that successor at its real timestamp would supply part of the execution outcome to the simulator and suppress this effect. Under equation 1, program-level error measures the timing model together with the demand its predictions induce.

Both branches route calls at release and replay tool delays after completion. Native dependency support or completion-driven extensions enforce the same rule. Appendix B gives the pseudocode and adaptations.

3.4 Simulator Integration

Adapters connect the common experiment to each simulator's execution model, translating inputs and collecting completion records. Native mechanisms implement the protocol where possible; extensions supply missing behavior. SplitwiseSim illustrates the boundary: Arena's completion callback releases a successor after its tool delay, while SplitwiseSim's scheduler and timing model determine when it completes. Small checks verify call counts, output totals, release timing, and replica routing. Table 6 describes each simulator's modifications.

Timing inputs remain simulator-specific: whole-batch measurements, operator composition, learned predictors, and analytical fallbacks retain different execution details. Replacing them with a shared model would change the systems being compared.

3.5 Evaluation Dimensions

The four RQs address different selection needs. **RQ1** identifies supported mechanisms and adaptations (Section 5.1). **RQ2** measures accuracy through mean program completion-time error (Section 5.2). **RQ3** compares accuracy with runtime, exposing the cost of exploring more configurations (Section 5.3). **RQ4** distinguishes preparation and resource costs from repeated execution (Section 5.4). Arena reports these dimensions separately; Section 4.5 defines their measurement boundaries.

4 Experimental Setup

4.1 Evaluated Simulators

We evaluate SplitwiseSim, DynoSim, APEX, TokenSim, LLMServingSim 2.0, and Frontier with their own timing models and profiles. Appendix B describes adaptations; Appendix F.2 details profiling inputs.

Cell	GPU	Model	Workload	Programs	TP	Instances
C0	B200	Llama-3.1-8B	SWE-LB	100	1	1
C1	B200	Llama-3.1-70B	SWE-bench	50	2	1
C2	B200	Mixtral-8x7B	BFCL	150	1	1
C3	B200	Llama-3.1-70B	SWE-bench	50	1	2
C4	RTX PRO 6000	Llama-3.1-8B	SWE-LB	100	1	1
C5	RTX PRO 6000	Llama-3.1-70B	SWE-bench	50	2	1
C6	RTX PRO 6000	Mixtral-8x7B	BFCL	150	2	1
C7	RTX PRO 6000	Llama-3.1-8B	SWE-bench	50	1	2

Table 1: Deployment matrix. Each row is evaluated at five program arrival rates. TP is per instance; GPU count is TP multiplied by instance count.

4.2 Experiment Matrix

Table 1 pairs eight deployments with fixed workloads, spanning Llama-3.1-8B/70B (Grattafiori et al., 2024), Mixtral-8x7B (Jiang et al., 2024), and B200 and RTX PRO 6000 GPUs. Each row is evaluated at 0.02, 0.04, 0.06, 0.08, and 0.10 programs/s. A setting is one row at one rate, giving 40 settings, 240 simulator predictions, and 40 real references. TP operates within an instance; replicas maintain independent state. C1/C3 provides the equal-GPU-count comparison in the introduction. The rate sweep changes demand while holding each row's programs and deployment fixed.

4.3 Workloads

We use software-agent and function-calling traces (Jimenez et al., 2024; Patil et al., 2025): SWE-LB (100 programs, 4,525 calls), SWE-bench (50 programs, 2,873 calls), and BFCL (150 programs, 1,370 calls). SWE-LB comes from leaderboard agent trajectories. Both branches replay fixed call inputs, output lengths, and tool delays. Program arrivals follow a Poisson process with seed 42, scaling the same draws across rates while preserving program order and contents. Later calls follow equation 1. BFCL prompts are retokenized for Mixtral; Appendix C gives provenance, preprocessing, and statistics.

4.4 Reference Serving

Real executions use stock vLLM (Kwon et al., 2023) `0.19.0+precompiled`, BF16 weights, prefix caching, and limits of 128 sequences and 16,384 batched tokens per instance. Both branches route round-robin at call release across replicas with independent caches. Measured per-instance KV budgets align capacity (Appendix C.2).

4.5 Metrics and Accounting

Program-level accuracy. Job completion time (JCT) spans prescribed program arrival to the final call's last output token, including queueing and intervening tool delays. Its mean captures the average time to finish a complete program under dependent execution. Every program contributes to the mean. For simulator s and setting c, absolute relative error is

$$E_{s,c} = 100\,\frac{|\widehat{J}_{s,c} - J_c|}{J_c}, \tag{2}$$

where J_c and $\widehat{J}_{s,c}$ are real and predicted JCT means over all programs. Aggregate error weights the 40 settings equally.

Runtime and tradeoffs. Runtime is wall time for a full-run adapter invocation, including initialization, conversion, and predictor fitting inside the invocation, but excluding cluster queueing and separate profiling. We report its median across the 40 settings. Within a setting, a result is nondominated if no other simulator has both no greater error and no greater runtime, with a strict improvement in either. Nondominated counts thus compare alternatives for the same experiment.

Preparation and resources. We report allocation-based profiling and execution costs separately. A reusable profile costing P contributes P/N per prediction over N compatible runs; unrecorded costs remain unspecified (Appendix F).

Capability	SW	DY	AP	TS	LS	FR
Matched KV capacity	◐	✓	✓	†	✓	✓
Prefix caching	×	◐	×	✓	✓	†
Chunked prefill	×	✓	×	×	✓	✓
Dependent call release	†	✓	†	†	✓	✓
Per-turn tool delays	†	✓	†	†	✓	†
Per-call replica routing	✓	✓	†	✓	✓	†
Program-directed KV retention	×	×	×	×	×	×
Cache- and load-aware routing	◐	✓	×	◐	◐	◐
History-aware scheduling	×	×	×	×	×	◐

Table 2: Agent-serving capabilities. SW: SplitwiseSim; DY: DynoSim; AP: APEX; TS: TokenSim; LS: LLMServingSim 2.0; FR: Frontier. ✓: supported; ◐: partial; ×: unestablished; †: Arena extension. Full matrix: Table 7.

5 RESULTS

We compare mechanism support (RQ1) and accuracy–cost tradeoffs (RQ2–RQ4) across 240 predictions and 40 real executions.

Results describe the pinned implementations with the disclosed Arena extensions, profiles, and configurations. Table 6 summarizes the per-simulator modifications.

5.1 RQ1: SUPPORT FOR AGENTIC-SERVING MECHANISMS

Only three of six simulators natively release calls according to program completion dependencies: DynoSim, LLMServingSim 2.0, and Frontier. SplitwiseSim, APEX, and TokenSim require completion-driven release extensions. Frontier additionally requires changes for recorded tool delays and routing at each release. Thus, accepting the same program trace does not by itself establish the same experiment. Table 2 summarizes agent-serving support, and Table 6 describes how the integrations enforce the common protocol.

A mechanism label does not specify its execution behavior. Prefix caching illustrates why support alone does not specify execution behavior. SplitwiseSim and APEX recompute shared context at every call. DynoSim reuses prompt blocks but not blocks generated by the preceding call, an important source of reuse when generation becomes part of a later prompt. Frontier requires a correction to its multi-round prefix lookup. Reuse also depends on placement: context retained at one replica is unavailable at another. These distinctions determine the prefill work presented to the timing model and the capacity left for competing programs. Matching KV capacity therefore aligns a resource constraint, not cache behavior.

Program replay is more widely supported than program-aware control. None of the examined integrations exposes explicit control over which program's context to retain during tool waits. Only DynoSim fully meets our criterion for routing based on both cache contents and server load; Frontier provides partial history-aware scheduling. A simulator can therefore execute a dependent trace without exposing the mechanisms needed to explore policies that coordinate an entire program. The numerical experiments use the fixed reference configuration in Section 4.4.

5.2 RQ2: ACCURACY AGAINST REAL SERVING PERFORMANCE

LLMServingSim 2.0 achieves the lowest mean absolute relative error (7.1%), but no simulator stays within 10% throughout the matrix. It meets that threshold in 30 of 40 settings, DynoSim in 24, and TokenSim in 17 (Table 3). The aggregate ranking differs from the frequency of best predictions: DynoSim has the smallest error in 16 settings, LLMServingSim in 15, TokenSim in eight, and APEX in one. Simulator selection therefore depends on where accuracy is needed, not only on the matrix-wide average.

Aggregate accuracy does not transfer uniformly across deployments. Figure 2 shows where aggregate rankings conceal different behavior. LLMServingSim's mean error is 2.1% for B200 70B TP2 (C1) but 21.3% for two B200 70B TP1 instances (C3), where it underpredicts at every arrival

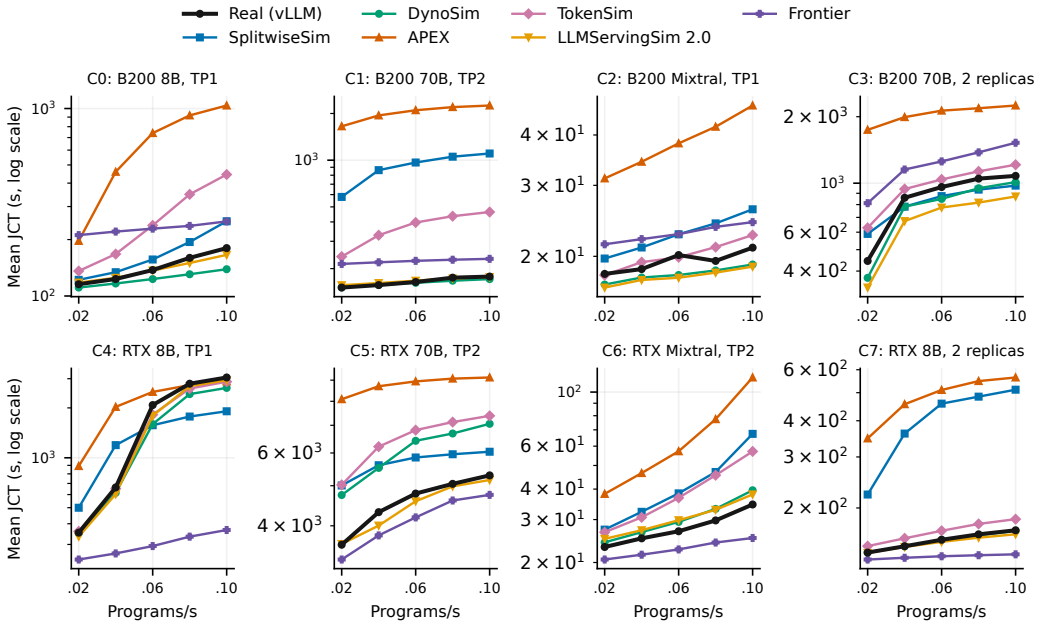


Figure 2: Mean program JCT versus arrival rate. Black: real vLLM; colors: simulators. Vertical axes are logarithmic and independently scaled. Deployments follow Table 1; RTX denotes RTX PRO 6000.

	JCT error (%)		Settings / 40			Runtime	
Simulator	Mean	Median	$\leq 10\%$	Best	Pareto	Median (s)	Sum (h)
SplitwiseSim	97.0	31.2	6	0	28	6.94	0.093
DynoSim	10.6	8.0	24	16	39	6.86	0.086
APEX	279.7	128.5	2	1	3	30.12	0.336
TokenSim	39.4	16.3	17	8	8	76.48	0.898
LLMServingSim 2.0	7.1	5.7	30	15	15	1,224.26	14.175
Frontier	33.8	28.6	4	0	5	1,498.68	27.930

Table 3: Accuracy and cost over 40 settings per simulator. Errors are absolute relative JCT errors. Best and Pareto count smallest-error and nondominated results within matched settings. Runtime is adapter wall time, excluding profiling.

rate. DynoSim averages 10.5% in C3 and 0.25% in the RTX two-instance configuration (C7), but 31.9% for RTX 70B TP2 (C5). TokenSim averages 4.3% on B200 Mixtral/BFCL (C2) and 5.6% on RTX 8B/SWE-LB (C4), compared with 123.9% on B200 70B/SWE-bench (C1). These differences make deployment-specific validation necessary even for simulators with favorable overall results.

The response to load matters as much as agreement at one point. For RTX 8B/SWE-LB (C4), real mean JCT grows from 353.55 s at 0.02 programs/s to 3,050.73 s at 0.10 programs/s. Frontier predicts only 243.33–366.64 s over the same range, reaching 88.0% error at the highest rate. At that rate, real mean time to first token is 26.18 s versus Frontier's 0.052 s, revealing a discrepancy before generation as well as in complete-program latency. Conversely, APEX's C4 error falls from 152.0% to 5.4% as load increases. Agreement at one operating point can therefore conceal a different response to demand. Appendix E provides supplementary token-latency measurements and an example where throughput agrees within 0.12% despite 72.6% JCT error.

5.3 RQ3: Predictive Accuracy Versus Simulation Speed

DynoSim is nondominated in 39 of 40 matched settings, combining 10.6% mean JCT error with a 6.86 s median runtime. LLMServingSim improves mean error to 7.1%, but its median runtime is 1,224.26 s, approximately $178\times$ longer. Their summed adapter times across the matrix are 0.086 and 14.175 hours. These aggregates establish the difference in repeated-evaluation cost; matched settings show what additional accuracy that cost buys.

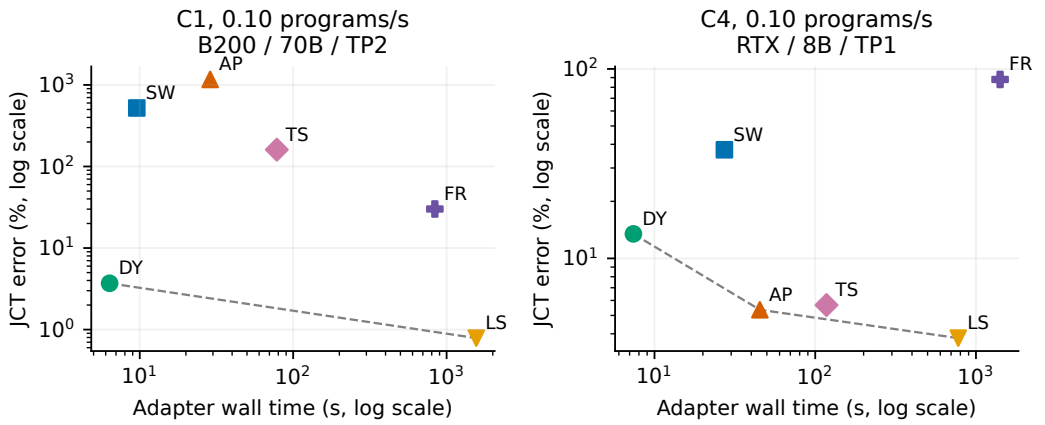


Figure 3: Accuracy–runtime tradeoffs at 0.10 programs/s: C1 (left) and C4 (right). Both axes are logarithmic; lower and left is better. Dashed lines connect nondominated observations. Abbreviations follow Table 2.

Simulator	Simulation CPU-h	Successful profiling GPU-h
SplitwiseSim	0.186	1.178
DynoSim	0.172	–
APEX	0.672	0.083
TokenSim	1.796	–
LLMServingSim 2.0	42.526	10.000
Frontier	111.718	10.173

Table 4: Simulation and profiling costs. CPU-hours cover 40 simulation runs per simulator; GPU-hours measure successful profiling. A dash indicates reuse of profiling data provided by the simulator.

In C1 at 0.10 programs/s (Figure 3), DynoSim obtains 3.70% error in 6.36 s, whereas LLMServingSim obtains 0.79% in 1,561.40 s. In C4 at the same rate, APEX offers an intermediate tradeoff: 5.36% error in 45.24 s, compared with DynoSim's 13.47% in 7.39 s and LLMServingSim's 3.80% in 776.05 s. A 10% error requirement in C4 would rule out DynoSim despite its favorable aggregate results; APEX would satisfy it at substantially lower runtime than LLMServingSim. The useful tradeoff is specific to the intended setting and accuracy requirement.

Nondominance alone is not an accuracy guarantee. SplitwiseSim has the smallest error in no setting, yet is nondominated in 28 because of its short runtimes; it falls within 10% error in only six. Frontier's five nondominated observations all occur in C5, while TokenSim's eight coincide with its eight smallest-error results. The frontier identifies tradeoffs; an experiment's error requirement determines which are useful.

5.4 RQ4: Preparation and Resource Requirements

All 240 simulator runs report zero execution GPU-hours, but every evaluated system relies on GPU-derived performance data. SplitwiseSim uses batch-latency profiles; DynoSim and TokenSim reuse AIConfigurator operator tables (Xu et al., 2026); APEX uses operator and collective tables; LLMServingSim reuses layerwise profiles; and Frontier fits predictors to measurements. Table 4 separates execution from profiling costs. Fast CPU execution is enabled by earlier measurement and model preparation, whose cost depends on the data available for the intended deployment.

The SplitwiseSim C0 example shows how preparation can outweigh repeated execution. One B200 8B profile job took 606 s, while five simulations together took 49.54 s. Across those predictions, profiling contributes 121.2 s per prediction versus 9.91 s average simulation time. Reusing the same profile for 40 compatible predictions would reduce its average preparation charge to 15.15 s. Compatibility, rather than sweep size alone, controls this benefit: identical replicas can share a per-instance profile, while a different model, GPU, or TP width may require new measurements. The practical cost of exploration therefore depends on both simulation speed and profile reuse.

6 DISCUSSION

Profile the states that agent execution visits. All six simulators use measured performance data, but retain different information when turning it into predictions. Whole-batch interpolation, operator composition, and learned prediction differ in both the unit measured and the state used to query it (Patel et al., 2023; Lin et al., 2025; Feng et al., 2026). SplitwiseSim's evaluated decode predictor retains batch size while collapsing context-length variation. LLMServingSim's attention lookup retains prefill-chunk size, prefill KV length, decode count, and decode KV length (Appendix F.2). These dimensions distinguish states that arise as agent contexts grow and calls compete for service. Alongside their 97.0% and 7.1% mean errors, this contrast motivates a hypothesis: retaining relevant state dimensions may be more valuable than collecting additional samples along a batch-size axis. Agent execution makes this especially important because predictions change subsequent demand and therefore the states queried next.

Shared measurements do not ensure shared fidelity. DynoSim and TokenSim reuse AIConfigurator data (Xu et al., 2026), yet average 10.6% and 39.4% error; in C1, their errors are 1.9% and 123.9%. Their integrations differ in timing composition, fallback, and execution modeling. Likewise, APEX's operator profiles and Frontier's learned predictors yield 279.7% and 33.8% mean error. The useful distinction is more specific than measured versus analytical or lookup versus learned prediction: which operations and contexts are covered, how timings are composed, and how scheduling and caching determine the work being timed.

Distinguish scale error from load-response error. To separate scale mismatch from response-shape mismatch, we scale each simulator's final JCT predictions to match one real measurement at 0.02 programs/s per deployment, then score only the other four rates. Across these 32 settings, DynoSim's mean error falls from 11.3% to 5.6%, and LLMServingSim's from 7.3% to 3.4%; SplitwiseSim and APEX retain 35.1% and 53.7% error (Appendix G). A scalar correction changes the baseline but preserves relative latency growth with load. This is a diagnostic of final JCT, including tool delays, and does not rerun the completion-dependent feedback.

Reporting which batch/KV states use measured, interpolated, extrapolated, or analytical timing would distinguish coverage gaps from composition errors. Auditing states visited during complete programs would target profiling effort and inform deployment choices.

7 CONCLUSION

AgentSimArena compares serving simulators against real agent execution through completion-dependent replay and matched deployment constraints. Across 240 predictions, LLMServingSim achieves the lowest aggregate error and DynoSim offers a faster alternative, but no simulator stays within 10% error throughout the matrix. Fidelity depends on the execution states represented by timing profiles and the response to load. Arena makes these accuracy–cost tradeoffs explicit for workload-specific simulator selection.

AI USE STATEMENT

We used AI-assisted tools to support manuscript editing, code development, and analysis checks. The authors reviewed the resulting text, code, and analyses and take responsibility for the paper's claims and conclusions.

REPRODUCIBILITY STATEMENT.

The benchmark code is available in our anonymous repository. Experimental settings and simulator adaptations are described in Section 4 and Appendix B.

REFERENCES

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. InferCept: Efficient intercept support for augmented large language model inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 81–95, 2024. URL https://proceedings.mlr.press/v235/abhyankar24a.html.

Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for LLM inference, 2024a. URL https://arxiv.org/abs/2405.05465v2.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, 2024b. URL https://www.usenix.org/conference/osdi24/presentation/agrawal.

Jaehong Cho, Minsu Kim, Hyunmin Choi, Guseul Heo, and Jongse Park. LLMServingSim: A HW/SW co-simulation infrastructure for LLM inference serving at scale, 2024. URL https://arxiv.org/abs/2408.05499v1.

Jaehong Cho, Hyunmin Choi, Guseul Heo, and Jongse Park. LLMServingSim 2.0: A unified simulator for heterogeneous and disaggregated LLM serving infrastructure, 2026. URL https://arxiv.org/abs/2602.23036.

Wei Da and Evangelia Kalyvianaki. LLM-emu:profile-driven native online emulation of LLM serving systems. In *Machine Learning for Computer Architecture and Systems 2026*, 2026. URL https://openreview.net/forum?id=9jrjM8ZQyn.

Yongming Ding, Rudy Pei, Hongkuan Zhou, Ryan Olson, Alec Flowers, and Vikram Sharma Mailthody. DynoSim: Simulating the pareto frontier. NVIDIA Technical Blog, May 2026. URL https://developer.nvidia.com/blog/dynosim-simulating-the-pareto-frontier/.

Yicheng Feng, Xin Tan, Yangtao Deng, Yimin Jiang, Yibo Zhu, and Hong Xu. Frontier: Towards comprehensive and accurate LLM inference simulation, 2026. URL https://arxiv.org/abs/2605.21312v2.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The Llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020. doi: 10.1109/ISCA45697.2020.00047. URL https://mkhairy.github.io/Docs/Accel-Sim.pdf.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. URL https://arxiv.org/abs/2309.06180.

Yi-Chien Lin, Woosuk Kwon, Ronald Pineda, and Fanny Nina Paravecino. APEX: An extensible and dynamism-aware simulator for automated parallel execution in LLM serving, 2025. URL https://arxiv.org/abs/2411.17651v2.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for LLM agents as general programs, 2025. URL https://arxiv.org/abs/2502.13965v1.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting, 2023. URL https://arxiv.org/abs/2311.18677.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale. 6 2018. doi: 10.1184/R1/6608078.v1. URL https://kilthub.cmu.edu/articles/journal_contribution/Open_Versus_Closed_A_Cautionary_Tale/6608078.

Feiyang Wu, Zhuohang Bian, Guoyang Duan, Tianle Xu, Junchi Wu, Teng Ma, Yongqiang Yao, Ruihao Gong, and Youwei Zhuo. TokenSim: Enabling hardware and software exploration for large language model inference systems, 2025. URL https://arxiv.org/abs/2503.08415v1.

Tianhao Xu, Yiming Liu, Xianglong Lu, Yijia Zhao, Xuting Zhou, Aichen Feng, Yiyi Chen, Yi Shen, Qin Zhou, Xumeng Chen, Ilya Sherstyuk, Haorui Li, Rishi Thakkar, Ben Hamm, Yuanzhe Li, Xue Huang, Wenpeng Wu, Anish Shanbhag, Harry Kim, Chuan Chen, and Junjie Lai. Aiconfigurator: Lightning-fast configuration optimization for multi-framework llm serving, 2026. URL https://arxiv.org/abs/2601.06288.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022. URL https://www.usenix.org/conference/osdi22/presentation/yu.

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. SGLang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, 2024. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Source	Modeling focus	Reported evidence
Vidur (Agrawal et al., 2024a)	Profile-based serving simulation	Conversation, summarization, and translation requests; instrumented vLLM on A100/H100; request latency and capacity.
LLMServingSim (Cho et al., 2024)	Serving and hardware co-simulation	ShareGPT-derived arrivals; throughput trends against GPU vLLM; heterogeneous accelerator studies.
Splitwise / SplitwiseSim (Patel et al., 2023)	Phase splitting and provisioning	Hardware characterization and production-derived traces; throughput, cost, and power exploration.
APEX (Lin et al., 2025)	Batching and parallel execution	vLLM/SGLang on H100/H200 clusters; predicted and measured execution-plan speedups.
TokenSim (Wu et al., 2025)	Scheduling and memory	Includes execution-time validation against DistServe on two A100 GPUs with fixed-length requests.
Frontier (Feng et al., 2026)	Co-located and disaggregated serving	Throughput and latency validation on a 16-H800 testbed.
LLM-Emu (Da & Kalyvianaki, 2026)	Wall-clock vLLM emulation	Repository timing comparisons for Qwen models on RTX 8000 and A40.
DynoSim (Ding et al., 2026)	Multi-turn sessions and routing	Documentation specifies completion-dependent turns and delays.

Table 5: Selected prior validation evidence. The LLMServingSim row describes the original publication; LLM-Emu and DynoSim rows summarize repository or documentation evidence.

Simulator	Arena modifications and configuration
SplitwiseSim	Added completion-driven call release: each successor enters the simulator after its predecessor completes and its tool delay elapses. The existing scheduler and timing model determine execution.
DynoSim	Configured native sessions with recorded inter-call delays, round-robin routing, and the reference per-instance KV budget.
APEX	Added a cooperative event loop that inserts dependent calls as they become ready, with round-robin routing across independent serving instances.
TokenSim	Added program processes that await call completion before releasing successors, independent replica execution, and an explicit per-worker KV-capacity override checked against available memory.
LLMServingSim 2.0	Used the native agentic router with recorded tool delays, round-robin placement, and measured per-instance capacity. Resolved the layerwise profiles for each deployment.
Frontier	Replaced configured tool waits with recorded per-call delays; cleared replica affinity to route each released call round-robin; and restricted prefix-cache lookup to complete blocks in the current prompt.

Table 6: Per-simulator changes and configuration used to execute Arena's common protocol.

A Prior Validation Evidence

Table 5 expands the validation discussion in Section 2. It records the workloads, reference systems, and metrics used in prior studies, distinguishing published validation evidence from the pinned implementations evaluated by Arena (Appendix B).

B Simulator Adaptations

Table 6 summarizes the changes made for Arena's evaluation. DynoSim and LLMServingSim use native dependent execution; their entries describe configuration rather than a new call-release mechanism. Timing models and profiling inputs are described in Appendix F.2.

Small integration checks verify completed-call counts, output-token totals, successor release times, replica routing, and cache-capacity translation.

The architecture audit identifies PDD support through SplitwiseSim's KV-transfer schedulers, DynoSim's disaggregated planner replay, TokenSim's prefill/decode worker pools, LLMServingSim's phase-transfer router, and Frontier's role-specific clusters. Frontier also exposes separate attention

```
for program p:
    schedule(time=p.arrival, event=release(p, 0))
release(p, k):
    backend.submit(p.calls[k], identity=(p.id, k))
on_complete(p, k, completion_time):
    record(p.id, k, completion_time)
    if k + 1 < len(p.calls):
        ready = completion_time + p.calls[k].tool_delay
        schedule(time=ready, event=release(p, k + 1))
```

Figure 4: Dependency-preserving replay, in backend-independent pseudocode. Scheduling uses the backend's logical or wall clock; routing occurs when a call is released. The evaluated multi-instance configurations use round-robin routing at release in both branches; Setup specifies the policy.

and FFN workers. PP is implemented in APEX execution plans, TokenSim's rank/stage model, LLMServingSim's pipeline scheduler, and Frontier's replica configuration. SplitwiseSim's timing model omits its PP field, yielding partial support; DynoSim's PP support is unestablished. DP denotes independent replicated serving.

MoE support denotes model or timing paths specific to expert computation. SplitwiseSim supplies whole-batch Mixtral measurements with generic request execution, yielding partial support. Batch scheduling denotes request selection for execution. TokenSim submits the whole uncached prompt before advancing to decoding, so its evaluated path lacks chunked prefill. SplitwiseSim's partial capacity entry reflects byte-level accounting rather than a block-cache model.

B.1 Replay and Adapter Interface

The adapter translates experiment inputs, invokes the backend, and normalizes its outputs through the following interface.

Operation	Contract
`supports(cell)`	Accept the deployment or return an explanatory reason.
`run(cell, policy, outdir, timeout)`	Check policy support, translate inputs, execute, and return a common result record.

B.2 Capability Evidence

The capability classifications use the following implementation behaviors and definitions.

TokenSim's capacity override is validated at cache initialization; Frontier's prefix-cache correction bounds hash lookup by the current prompt length. Table 6 summarizes these changes. DynoSim's partial prefix-caching entry reflects reuse of prompt blocks but not preceding-call decode blocks.

B.3 Capability Reporting Schema

Each mechanism is described using the fields below. Entries distinguish native support, Arena extensions, unsupported behavior, and unverified capabilities.

Persistent identity links successive calls through a session or program identifier. Program-directed KV retention explicitly protects or releases a program's context across a tool wait. Cache- and load-aware routing uses both destination cache residency and load; load-only or sticky placement receives partial credit. History-aware scheduling uses earlier calls or accumulated program progress. Frontier's round-sensitive priorities and specialized scheduler state receive partial credit.

The policy audit covers DynoSim's KV-router mode; load-based placement in SplitwiseSim, TokenSim, and LLMServingSim; and Frontier's initial least-loaded placement with session affinity. Program-specific KV retention is unsupported across the six integrations. The numerical experiments use the fixed routing and serving configuration in Section 4.4.

Capability	SW	DY	AP	TS	LS	FR
Serving architecture and parallelism						
Prefill/decode disagg. (PDD)	✓	✓	×	✓	✓	✓
Attention/FFN disagg. (AFD)	×	×	×	×	×	✓
Tensor parallelism (TP)	✓	✓	✓	✓	✓	✓
Pipeline parallelism (PP)	◐	×	✓	✓	✓	✓
DP (replicated serving)	✓	✓	✓	✓	✓	✓
MoE modeling	◐	✓	✓	✓	✓	✓
Runtime mechanisms						
Batch scheduling	✓	✓	✓	✓	✓	✓
Matched KV capacity	◐	✓	✓	†	✓	✓
Prefix caching	×	◐	×	✓	✓	†
Chunked prefill	×	✓	×	×	✓	✓
Agentic execution						
Persistent program identity	†	✓	†	†	✓	✓
Dependent call release	†	✓	†	†	✓	✓
Per-turn tool delays	†	✓	†	†	✓	†
Per-call replica routing	✓	✓	†	✓	✓	†
Policies across calls						
Program-directed KV retention	×	×	×	×	×	×
Cache- and load-aware routing	◐	✓	×	◐	◐	◐
History-aware scheduling	×	×	×	×	×	◐

Table 7: Simulator capability coverage. SW: SplitwiseSim; DY: DynoSim; AP: APEX; TS: TokenSim; LS: LLMServingSim 2.0; FR: Frontier. ✓: supported; ◐: partial; ×: support not established; †: Arena extension. Architecture and policy rows audit capabilities beyond the accuracy matrix. Definitions and source evidence appear in Appendix B.2.

Statistic	SWE-LB	SWE-bench	BFCL (Mixtral)
Programs	100	50	150
Calls	4,525	2,873	1,370
Calls/program, mean	45.25	57.46	9.13
Calls/program, median	43	47	8
Calls/program, range	21–125	10–185	3–21
Input tokens/call, median	12,302	11,115	5,925
Input tokens/call, p95	28,522.6	33,690.4	32,742.6
Output tokens/call, median	130	52	26
Output tokens/call, p95	1,198.4	351.2	83.6
Tool delay (s), mean	1.188	1.114	1.492
Tool delay (s), median	0.157	0.200	1.182
Tool delay (s), p95	5.023	3.667	3.472

Table 8: Trace statistics. Token statistics weight calls equally; tool-delay statistics cover only gaps between calls, excluding terminal fields. Percentiles use linear interpolation. Identical program contents are reused at every arrival rate.

Field	Meaning
Status	Native, Arena extension, unsupported, or unverified.
Scope	Backend revision and applicable deployment constraints.
Semantics	Implemented behavior and approximations.
Evidence	Implementation behavior, validation check, or available counter.

C Workload and Reference Configuration Details

C.1 Trace Composition and Arrival Construction

The workload audit checks the invariant used in Section 4.3: the rate sweep changes initial arrivals, not prescribed program work. Table 8 characterizes the three trace subsets at five rates. Removing the arrival field yields identical content hashes across rates within each subset. Every initial timestamp matches cumulative draws from Python's `random.Random(42).expovariate`, rounded to integer nanoseconds; the same draws are scaled across rates.

The workload inputs are the fixed `swelb100`, `swebench50`, and `bfcl150` subsets of the collected performance-trace corpus. Table 9 identifies their collection models and source collections. The

Subset	Collection model	Source and selection
SWE-LB	Claude Sonnet 4.5 (100 programs)	Public leaderboard trajectories, high reasoning effort; first 100 records of the converted collection. Agent version: mini-swe-agent 2.1.0.
SWE-bench	Qwen3-Coder (5); Qwen3.6-Plus (45)	First 50 records of the saved collection; model identities recorded per program.
BFCL	Qwen3.6-Plus (150)	Selected programs from the BFCL v4 collection.

Table 9: Collection provenance, distinct from the models used for serving replay. Source-file hashes and selected program IDs pin all three subsets; immutable task-dataset revisions were not recorded in the collection metadata.

Cell	Instances	TP per instance	KV tokens per instance
C0	1	1	1,175,248
C1	1	2	593,360
C2	1	1	571,504
C3	2	1	83,312
C4	1	1	561,664
C5	1	2	115,824
C6	1	2	651,200
C7	2	1	561,664

Table 10: Measured KV-cache budgets used across all five rates of each deployment. Capacity is per serving instance, not a sum over TP ranks or independent replicas.

benchmark papers identify the task families (Jimenez et al., 2024; Patil et al., 2025); they are not the source of the timing distributions in Table 8. Replay uses recorded prompts and requested output lengths rather than regenerating the agent's choices: a later prompt remains the recorded input even when real inference produces different text.

All 300 selected programs were matched to their collection records, verifying replay token IDs, output lengths, and tool delays against the saved subsets. Conversion orders calls by turn index, converts integer millisecond tool delays to nanoseconds, and sets the terminal delay to zero. Where the source replay-output count is nonpositive, it uses the collection-output count clipped to $[1, 2048]$ tokens; this applies to 4,525 SWE-LB, 2,067 SWE-bench, and 8 BFCL calls. SWE-bench also has 1,109 changed tool-name labels; these do not change its verified token or delay fields.

BFCL preprocessing converts Llama-tokenized prompts to Mixtral and left-truncates inputs where needed to fit the 32,768-token combined input/output limit, preserving output counts and tool delays. Terminal tool delays are zero in every evaluated file.

C.2 Reference Engine and Measured KV Budgets

All 40 reference configurations record vLLM `0.19.0+precompiled`, BF16 weights, automatic KV-cache dtype, and 16-token blocks. Effective engine records confirm 128 sequences and 16,384 batched tokens per instance, with 65,536-token Llama contexts and 32,768-token Mixtral contexts. Commands enable prefix caching; the stock engine determines admission, batching, and cache eviction. Recorded effective instances use asynchronous scheduling. Independent replicas use round-robin routing at call release, without cross-replica cache sharing.

Capacity capture reads the initialized engine's GPU-block count and block size, with usable tokens equal to their product. This budget is fixed before collecting reference JCT. Subsequent replays pass the corresponding GPU-block override and validate the resulting effective capacity, rather than estimating it from nominal device memory. Engine readiness checks and startup complete before the workload clock begins. Reference validation requires the expected number of programs, unique call identities, prescribed output-token totals, and complete lifecycle timestamps. JCT ends at the final call's last output token; it includes real submission lateness relative to the prescribed initial arrival, but excludes engine startup and any post-program bookkeeping.

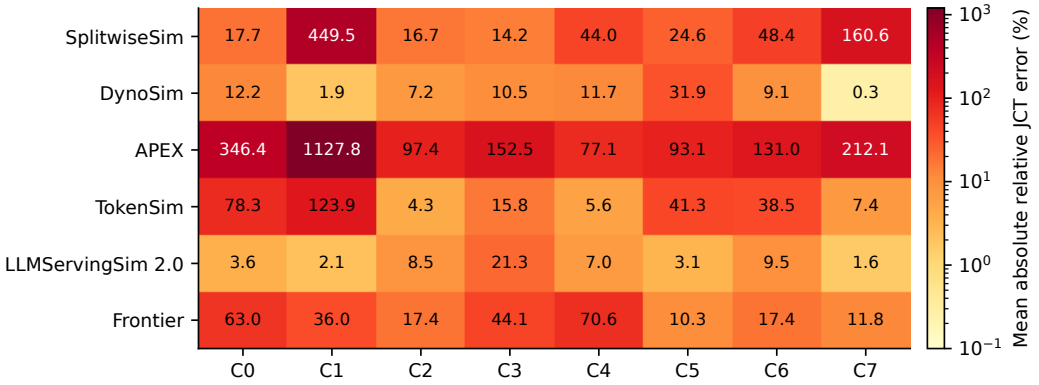


Figure 5: Mean absolute relative JCT error (%) over five arrival rates per deployment. Annotations are the actual percentage errors; color uses a logarithmic scale. Each row covers the same 40 settings.

Simulator	C0	C1	C2	C3	C4	C5	C6	C7	Total
SplitwiseSim	4	1	5	5	0	3	5	5	28
DynoSim	5	5	5	4	5	5	5	5	39
APEX	0	0	0	0	3	0	0	0	3
TokenSim	0	0	4	2	2	0	0	0	8
LLMServingSim 2.0	5	2	0	0	1	5	2	0	15
Frontier	0	0	0	0	0	5	0	0	5

Table 11: Nondominated observations out of five rates per cell and 40 settings overall.

D Full Results

The evaluation contains 40 real references and 240 simulator observations. Tables 12–19 report mean program JCT and adapter wall time for every setting, using the deployment identifiers in Table 1. Errors are computed against the real mean JCT at the same deployment and rate, then averaged equally across the 40 settings. Runtime medians use those same settings; Pareto comparisons are performed within each setting.

Full-precision JCT and runtime records are joined by source identity; all 280 observations pass this consistency check.

D.1 Accuracy Across Deployments

Figure 5 averages the five setting-level percentage errors within each cell. LLMServingSim has the lowest cell-average error in C0 and C5; DynoSim in C1, C3, C6, and C7; and TokenSim in C2 and C4. These cell averages complement the per-setting best counts in Table 3: they summarize a different level of aggregation.

D.2 Nondominated Settings

Table 11 counts observed nondominated error–runtime pairs within each deployment. Each decision compares all six simulators at the same arrival rate and deployment. A pair is dominated if another has no larger error and no larger wall time, with at least one strict improvement. Multiple simulators may be nondominated, so column totals can exceed five. The counts use full-precision recorded values without an uncertainty margin; they are descriptive comparisons of the selected runs.

D.3 Complete JCT and Runtime Measurements

JCT measures the delay experienced by a program; wall time measures simulator execution. Each table reports five arrival-rate settings.

System / metric	0.02	0.04	0.06	0.08	0.10
Mean program JCT (seconds)					
Real (vLLM)	115.57	123.19	137.55	159.75	180.10
SplitwiseSim	121.94	133.95	156.57	194.03	250.43
DynoSim	110.93	116.58	123.16	130.63	138.95
APEX	196.69	459.50	739.45	918.67	1037.46
TokenSim	135.87	167.39	237.99	348.47	444.56
LLMServingSim 2.0	117.91	125.00	136.70	149.96	165.76
Frontier	211.23	220.58	228.73	236.58	249.95
Adapter wall time (seconds)					
SplitwiseSim	8.38	8.77	9.22	10.44	12.73
DynoSim	19.41	14.87	12.17	10.57	9.59
APEX	58.23	43.17	42.00	42.30	40.94
TokenSim	155.87	134.65	121.93	118.39	124.43
LLMServingSim 2.0	4970.42	2939.85	1954.54	1627.60	1256.49
Frontier	3041.13	2053.04	1886.65	1434.41	1354.95

Table 12: C0: B200 / 8B / TP1 / SWE-LB. Columns are program arrival rates in programs/s. Each entry is one selected run.

System / metric	0.02	0.04	0.06	0.08	0.10
Mean program JCT (seconds)					
Real (vLLM)	150.99	156.64	163.97	174.85	177.70
SplitwiseSim	579.68	862.49	966.01	1053.21	1104.33
DynoSim	151.26	156.91	162.00	167.11	171.13
APEX	1652.86	1943.23	2098.46	2198.51	2250.84
TokenSim	238.98	328.31	396.58	435.36	463.26
LLMServingSim 2.0	156.34	161.39	167.16	172.67	176.31
Frontier	214.43	219.67	224.47	228.19	231.21
Adapter wall time (seconds)					
SplitwiseSim	7.00	7.68	8.25	8.70	9.57
DynoSim	7.34	6.56	6.45	6.20	6.36
APEX	31.45	30.39	28.80	28.67	28.77
TokenSim	79.47	77.29	75.83	77.03	78.23
LLMServingSim 2.0	2656.80	2087.29	1843.49	1563.73	1561.40
Frontier	1314.41	1103.98	863.66	894.08	838.31

Table 13: C1: B200 / 70B / TP2 / SWE-bench. Columns are program arrival rates in programs/s. Each entry is one selected run.

System / metric	0.02	0.04	0.06	0.08	0.10
Mean program JCT (seconds)					
Real (vLLM)	18.09	18.59	20.15	19.48	21.02
SplitwiseSim	19.75	21.03	22.68	24.14	26.17
DynoSim	17.02	17.72	17.97	18.45	19.08
APEX	31.21	34.32	38.13	41.92	47.33
TokenSim	17.96	19.35	19.89	21.08	22.58
LLMServingSim 2.0	16.73	17.47	17.70	18.22	18.86
Frontier	21.42	22.07	22.70	23.66	24.31
Adapter wall time (seconds)					
SplitwiseSim	3.13	3.13	3.08	3.10	3.06
DynoSim	6.07	5.92	5.94	5.88	5.86
APEX	14.24	14.13	13.91	13.74	13.27
TokenSim	43.08	43.44	41.53	41.71	43.23
LLMServingSim 2.0	577.18	365.07	329.09	436.14	271.82
Frontier	5666.26	5879.26	5767.44	5115.46	4797.81

Table 14: C2: B200 / Mixtral / TP1 / BFCL. Columns are program arrival rates in programs/s. Each entry is one selected run.

System / metric	0.02	0.04	0.06	0.08	0.10
Mean program JCT (seconds)					
Real (vLLM)	444.51	859.05	960.60	1048.66	1079.26
SplitwiseSim	589.27	782.83	873.97	935.05	975.09
DynoSim	372.67	783.19	848.89	948.91	1011.06
APEX	1743.67	1991.04	2130.43	2184.92	2247.31
TokenSim	627.88	940.64	1039.84	1131.34	1212.33
LLMServingSim 2.0	336.85	673.57	775.86	816.82	870.74
Frontier	811.97	1153.25	1254.93	1379.08	1524.08
Adapter wall time (seconds)					
SplitwiseSim	5.98	6.13	6.19	6.21	6.32
DynoSim	7.43	7.15	6.90	6.82	7.04
APEX	33.63	30.21	29.37	30.02	27.70
TokenSim	72.45	69.23	68.57	69.86	69.65
LLMServingSim 2.0	2095.78	1334.05	1447.60	1192.03	1867.59
Frontier	1781.90	1490.99	1611.92	1450.93	1710.34

Table 15: C3: B200 / 70B / 2 instances / SWE-bench. Columns are program arrival rates in programs/s. Each entry is one selected run.

System / metric	0.02	0.04	0.06	0.08	0.10
Mean program JCT (seconds)					
Real (vLLM)	353.55	662.22	2079.94	2798.43	3050.73
SplitwiseSim	499.55	1192.42	1571.77	1773.34	1908.31
DynoSim	350.20	614.46	1596.25	2422.15	2639.85
APEX	891.04	2027.68	2496.53	2744.97	2887.36
TokenSim	360.09	651.09	1820.89	2616.14	2877.69
LLMServingSim 2.0	334.57	603.26	1794.31	2706.05	2934.95
Frontier	243.33	264.78	293.35	334.31	366.64
Adapter wall time (seconds)					
SplitwiseSim	10.91	22.00	24.68	26.11	27.22
DynoSim	10.42	7.72	7.27	7.19	7.39
APEX	52.22	46.47	45.82	45.02	45.24
TokenSim	137.69	125.18	121.51	118.82	117.52
LLMServingSim 2.0	1370.99	701.63	618.53	373.48	776.05
Frontier	2967.88	2076.82	1711.98	1506.38	1413.28

Table 16: C4: RTX / 8B / TP1 / SWE-LB. Columns are program arrival rates in programs/s. Each entry is one selected run.

System / metric	0.02	0.04	0.06	0.08	0.10
Mean program JCT (seconds)					
Real (vLLM)	3600.12	4315.69	4786.42	5052.28	5296.35
SplitwiseSim	5007.66	5604.31	5851.07	5957.54	6041.54
DynoSim	4747.41	5521.50	6420.65	6687.56	7058.04
APEX	8093.79	8699.86	8937.92	9077.26	9134.64
TokenSim	5023.08	6214.55	6814.46	7126.59	7378.84
LLMServingSim 2.0	3614.68	4004.75	4586.20	4981.61	5162.25
Frontier	3315.14	3791.52	4191.26	4605.11	4751.60
Adapter wall time (seconds)					
SplitwiseSim	7.22	7.46	7.46	7.51	7.45
DynoSim	6.45	6.25	6.27	6.09	6.23
APEX	31.61	31.20	30.99	30.69	31.01
TokenSim	76.11	76.84	75.73	75.96	72.98
LLMServingSim 2.0	1551.37	1841.04	1532.95	1568.43	1585.77
Frontier	1258.89	1205.66	1216.58	1151.39	1110.96

Table 17: C5: RTX / 70B / TP2 / SWE-bench. Columns are program arrival rates in programs/s. Each entry is one selected run.

System / metric	0.02	0.04	0.06	0.08	0.10
Mean program JCT (seconds)					
Real (vLLM)	23.15	25.12	26.86	29.75	34.59
SplitwiseSim	27.28	32.28	38.42	47.01	67.39
DynoSim	24.19	26.64	29.35	33.18	39.56
APEX	38.22	46.54	57.17	77.40	114.76
TokenSim	26.61	30.64	36.82	45.60	57.10
LLMServingSim 2.0	25.00	27.13	29.82	32.88	38.03
Frontier	20.57	21.53	22.63	24.15	25.20
Adapter wall time (seconds)					
SplitwiseSim	3.34	3.37	3.43	3.45	3.33
DynoSim	6.17	5.91	5.89	5.69	5.60
APEX	18.35	17.18	16.88	15.87	15.32
TokenSim	47.12	45.68	44.57	43.23	41.42
LLMServingSim 2.0	1153.89	631.98	545.81	482.63	458.73
Frontier	6439.19	6375.01	5545.19	5577.12	5124.50

Table 18: C6: RTX / Mixtral / TP2 / BFCL. Columns are program arrival rates in programs/s. Each entry is one selected run.

System / metric	0.02	0.04	0.06	0.08	0.10
Mean program JCT (seconds)					
Real (vLLM)	140.33	147.51	155.34	162.19	167.49
SplitwiseSim	222.53	361.24	458.31	484.48	511.99
DynoSim	140.39	147.48	155.17	161.20	166.70
APEX	347.56	456.08	510.90	548.25	564.15
TokenSim	147.69	157.11	166.84	176.19	182.96
LLMServingSim 2.0	139.92	146.68	152.77	157.94	162.27
Frontier	132.76	134.61	136.25	137.37	138.35
Adapter wall time (seconds)					
SplitwiseSim	6.46	6.61	6.87	6.78	6.83
DynoSim	9.65	9.00	8.59	8.20	8.21
APEX	30.47	27.28	27.33	27.76	27.86
TokenSim	86.80	83.84	80.69	78.35	77.25
LLMServingSim 2.0	1125.81	680.50	556.66	510.64	586.41
Frontier	1385.94	1241.33	1068.41	1213.39	899.40

Table 19: C7: RTX / 8B / 2 instances / SWE-bench. Columns are program arrival rates in programs/s. Each entry is one selected run.

E LATENCY AND THROUGHPUT DIAGNOSTICS

Throughput agreement can hide program-latency error. At 0.02 programs/s on B200 Mixtral/BFCL (C2), APEX predicts 8.480 output tokens/s versus 8.490 measured, a difference of less than 0.12%, yet predicts 31.21 s mean JCT versus 18.09 s, an error of 72.6%. Throughput is computed over the interval from first arrival to last completion, which includes the prescribed arrival spacing. Under sparse arrivals, this interval can change little even when individual programs take substantially longer. The example illustrates why matching aggregate throughput does not ensure accurate agent completion times.

Table 20 supports the load-response and throughput examples in Section 5.2, reporting the endpoint arrival rates, 0.02 and 0.10 programs/s, for all eight deployments. These per-call means and whole-run rates supplement the primary program-level accuracy metric. TTFT measures the interval from a call's arrival to its first output token. For a call with $N > 1$ output tokens, time per output token (TPOT) is

$$\mathrm{TPOT} = \frac{t_{\mathrm{last}} - t_{\mathrm{first}}}{N - 1}. \tag{3}$$

We report per-call means and exclude one-token calls from TPOT. Output throughput divides the total output-token count by the time from the first arrival to the last completion. It is a whole-run rate, not a token-generation rate measured only while the GPU is busy.

TTFT and TPOT are unavailable for APEX and for Frontier in C5; these entries are marked with a dash.

Table 20: Per-call latency and output throughput at the two endpoint arrival rates. TTFT is in milliseconds; TPOT is in milliseconds per output-token interval; throughput is output tokens/s. A dash denotes unavailable timing.

Cell	System	0.02 programs/s			0.10 programs/s		
		TTFT	TPOT	Tokens/s	TTFT	TPOT	Tokens/s
C0	Real (vLLM)	24.60	4.42	274.58	27.94	8.96	1180.57
C0	SplitwiseSim	171.95	4.41	274.09	414.48	12.60	1105.04
C0	DynoSim	14.96	4.13	274.98	18.64	6.08	1198.61
C0	APEX	–	–	270.59	–	–	513.27
C0	TokenSim	82.02	5.71	273.59	106.46	27.46	936.76
C0	LLMServingSim 2.0	15.38	4.63	274.41	20.91	7.98	1181.48
C0	Frontier	20.14	11.30	265.36	22.50	14.03	1025.88
C1	Real (vLLM)	61.11	15.69	125.17	68.45	20.80	250.89
C1	SplitwiseSim	2021.86	74.74	78.46	3059.29	172.29	84.43
C1	DynoSim	55.02	15.99	125.07	63.10	19.91	255.64
C1	APEX	–	–	41.62	–	–	42.48
C1	TokenSim	329.72	29.54	110.54	458.39	75.31	151.13
C1	LLMServingSim 2.0	48.24	16.99	124.48	56.62	20.86	247.19
C1	Frontier	65.59	27.75	111.74	71.59	31.38	181.25
C2	Real (vLLM)	283.65	7.54	8.49	352.75	12.79	41.22
C2	SplitwiseSim	463.25	9.07	8.49	781.38	19.77	40.90
C2	DynoSim	236.35	6.55	8.49	313.00	10.37	41.28
C2	APEX	–	–	8.48	–	–	39.44
C2	TokenSim	337.68	6.66	8.49	523.62	14.94	41.12
C2	LLMServingSim 2.0	202.70	6.69	8.49	273.69	10.75	41.30
C2	Frontier	203.96	18.33	8.49	307.67	23.87	41.16
C3	Real (vLLM)	2493.46	39.36	80.74	11392.88	63.19	81.46
C3	SplitwiseSim	4211.86	52.52	77.10	8873.79	80.50	90.88
C3	DynoSim	1793.35	36.96	85.77	10878.52	61.45	84.84
C3	APEX	–	–	39.01	–	–	40.05
C3	TokenSim	4699.85	52.58	70.86	13481.71	68.64	74.80
C3	LLMServingSim 2.0	1347.79	35.67	88.94	8886.97	56.06	96.41
C3	Frontier	6108.07	81.05	62.23	16820.60	100.58	61.39
C4	Real (vLLM)	81.55	21.12	260.86	26183.35	126.13	260.46
C4	SplitwiseSim	952.32	28.44	258.68	16347.32	78.62	352.33
C4	DynoSim	73.39	20.95	261.91	21891.35	112.76	285.66
C4	APEX	–	–	238.46	–	–	248.91
C4	TokenSim	108.62	21.56	260.84	25064.92	117.87	271.88

Cell	System	0.02 programs/s			0.10 programs/s		
		TTFT	TPOT	Tokens/s	TTFT	TPOT	Tokens/s
C4	LLMServingSim 2.0	64.39	19.89	261.41	24860.83	123.29	265.71
C4	Frontier	38.22	13.55	262.84	51.63	22.23	942.66
C5	Real (vLLM)	39103.59	224.34	22.67	60662.67	313.46	21.81
C5	SplitwiseSim	56500.83	349.41	19.43	72979.41	364.81	19.52
C5	DynoSim	54558.14	285.78	18.26	84511.96	422.08	17.16
C5	APEX	–	–	12.68	–	–	12.69
C5	TokenSim	58132.09	288.49	17.19	88830.15	423.76	16.73
C5	LLMServingSim 2.0	40674.80	223.70	22.68	60922.29	310.60	22.23
C5	Frontier	–	–	23.96	–	–	23.15
C6	Real (vLLM)	503.69	14.88	8.49	779.21	38.71	40.24
C6	SplitwiseSim	888.88	19.07	8.49	3022.40	77.34	38.37
C6	DynoSim	637.67	15.38	8.49	1330.35	44.15	39.65
C6	APEX	–	–	8.47	–	–	35.86
C6	TokenSim	788.14	17.89	8.49	2074.57	72.76	37.98
C6	LLMServingSim 2.0	513.72	19.88	8.49	1021.58	45.98	39.91
C6	Frontier	192.31	16.73	8.49	277.60	27.13	41.10
C7	Real (vLLM)	63.53	13.50	125.98	76.49	18.24	259.87
C7	SplitwiseSim	832.05	20.93	118.87	2312.71	77.14	151.89
C7	DynoSim	65.13	13.63	125.83	74.08	18.37	266.60
C7	APEX	–	–	104.78	–	–	132.76
C7	TokenSim	101.64	14.48	125.12	112.67	21.30	251.67
C7	LLMServingSim 2.0	53.85	13.58	125.96	63.75	17.65	268.43
C7	Frontier	38.37	12.57	127.10	42.21	13.59	295.77

Simulator	Performance inputs	Cost not established
SplitwiseSim	Model/GPU/TP-specific batch-latency databases	Earlier B200 70B TP1 and RTX 70B TP2 profiling; full retry history.
DynoSim	Reused AIConfigurator operator database	Original collection of the reused database.
APEX	GPU-specific operator and collective tables	Operator tables and profiling preceding the measured communication job.
TokenSim	Imported AIConfigurator tables with analytical fallback	Original collection of the reused database.
LLMServingSim 2.0	Reused AgentServingSim dense, per-sequence, attention, and MoE layerwise tables; author-reported 10 GPU-h	Per-job breakdown of the reported total.
Frontier	Operator, attention, and communication measurements; CPU predictor fitting	Earlier profiles, other retries, and separately attributable predictor-fitting cost.

Table 21: Performance-data provenance and missing preparation costs. Data reuse describes the evaluated integrations. DynoSim and TokenSim share a timing-data source.

Simulator	Measurement unit and conditioning	Prediction and reuse
SplitwiseSim	Whole-batch prefill and decode timings, keyed by model/GPU/TP. Prefill queries retain aggregate prompt tokens; evaluated decode queries retain batch size.	Reused or newly collected vLLM batch profiles; median aggregation and one-dimensional linear interpolation/extrapolation. Mixed steps use a prefill-based estimate with a fixed multiplier.
DynoSim	AIConfigurator operator measurements for GEMM, attention, MoE, and collectives; model/backend-specific composition.	Reused database through the evaluated AIC timing provider; operator lookup/interpolation with backend-specific timing corrections.
APEX	GPU-specific GEMM and attention tables plus collective/message-size tables.	Operator interpolation and execution-graph composition; communication extrapolation for large messages. Evaluated BF16 workloads use upstream half-precision timing tables.
TokenSim	Imported AIConfigurator operator tables; keys include matrix shapes, attention batch/context dimensions, and collective message sizes.	Table-first lookup, interpolation, bounded extrapolation, then device-family analytical fallback; sums layer/operator estimates.
LLMServingSim 2.0	Dense-token, per-sequence, attention, and MoE tables. Attention retains prefill chunk, prefill KV, decode count, and decode KV dimensions.	Reused AgentServingSim profiles; linear dense/per-sequence interpolation and four-dimensional log-linear attention interpolation, with sparse-grid handling and skew correction.
Frontier	Model/TP-conditioned operator and attention measurements, including mixed-batch attention profiles.	Random-forest regression with fitted-predictor caching; evaluated communication path selects the analytical backend.

Table 22: Measurement granularity, prediction methods, and profile reuse in the evaluated integrations.

F Preparation and Resource Accounting

F.1 Performance-Data Reuse

Table 21 identifies each simulator's performance inputs and the preparation costs outside the reported totals. All six reuse prior measurements.

F.2 Timing Models and Profiling Granularity

Table 22 supplies the structural evidence for Section 6: what is measured, which execution-state dimensions are retained, and how measurements become predictions. It distinguishes data provenance from interpolation, regression, composition, and analytical fallback in the evaluated integrations. This makes profile compatibility and representation inspectable alongside end-to-end error.

Measurement boundaries and feature retention. SplitwiseSim's collection script times isolated synchronous vLLM batches with prefix caching and chunked prefill disabled. It measures one-token generation for prefill and estimates decode time from the difference between longer and one-token generations. These serving-call timings include overhead beyond individual kernels. Its decode predictor groups measurements by batch size, taking the median over prompt-length variation. LLMServingSim instead composes layerwise timings and queries attention using the current prefill/decode and KV state. Frontier fits regressors to operator measurements, including kernel-only measurement families. Consequently, profile values are not interchangeable even when model and GPU names match: their timing boundaries and retained state differ.

System	0.02	0.04	0.06	0.08	0.10	Wrong / 5
Real (vLLM)	2.944	5.484	5.859	5.997	6.073	–
SplitwiseSim	1.017	0.908	0.905	0.888	0.883	4
DynoSim	2.464	4.991	5.240	5.678	5.908	0
APEX	1.055	1.025	1.015	0.994	0.998	2
TokenSim	2.627	2.865	2.622	2.599	2.617	0
LLMServingSim 2.0	2.155	4.173	4.642	4.730	4.939	0
Frontier	3.787	5.250	5.591	6.043	6.592	0

Table 23: Predicted and measured C3/C1 mean-JCT ratios from the selected runs. Both deployments use two B200 GPUs. Ratios below one reverse the observed ordering.

F.3 Amortization and Measurement Boundaries

For N compatible predictions sharing preparation cost P, the average preparation charge is P/N, and the average serial wall-time cost is

$$\overline{T}(N) = \frac{P_{\text{wall}}}{N} + \frac{1}{N}\sum_{i=1}^{N} T_i. \tag{4}$$

GPU-hours and CPU-hours are reported separately; they are not added as interchangeable costs. In the measured SplitwiseSim C0 example, $P_{\text{wall}} = 606\,\text{s}$ for the profile job and $\sum_{i=1}^{5} T_i = 49.54\,\text{s}$. This gives 131.11 s per prediction under serial accounting, of which 121.20 s is the preparation charge. Holding that one-time charge fixed gives 15.15 s per prediction over 40 compatible uses. The latter value is an amortized preparation charge, not a measurement of a new 40-run campaign.

The adapter-time records cover successful full runs and exclude queueing, separate smoke jobs, failed attempts, and separately executed profiling. Initialization, conversion, and predictor fitting are included if they occur within the adapter's timing boundary. CPU-hours are allocation/assumption times adapter wall duration and cover all 40 settings for each simulator. LLMServingSim uses three allocated CPUs per selected run, giving $3 \times 14.17535 = 42.526$ CPU-hours. Execution GPU-hours are zero for all 240 simulator runs. Peak RSS, actual CPU utilization, and independent cold/warm initialization measurements are unavailable.

G Decision and Calibration Diagnostics

These analyses connect the practical deployment example in Sections 1 and 5.2 to the calibration diagnostic in Section 6. Both use the frozen JCT observations without new simulator executions. The first checks the predicted ordering of two deployment bundles; the second asks how much error remains after matching one operating point.

G.1 Deployment Choice at a Fixed GPU Count

C1 and C3 both use two B200 GPUs with Llama-3.1-70B and the same SWE-bench trace at each arrival rate. The ten reference configuration records confirm matching bfloat16 precision, KV-cache dtype, block size, per-instance sequence and batch-token limits, context limit, and vLLM version. C1 uses TP2 with one instance and 593,360 KV tokens; C3 uses TP1 with two instances and 83,312 KV tokens per instance. Thus this is a comparison of deployment bundles with equal GPU count, not an ablation isolating parallelism from memory capacity or routing.

Table 23 reports $J_{\text{C3}}/J_{\text{C1}}$. A ratio below one recommends replicas; a ratio above one recommends TP2. Real execution favors TP2 at all five rates; the wrong-choice count records disagreement with this ordering. At the largest rate, selecting replicas from SplitwiseSim's prediction would incur $1{,}079.26/177.70 = 6.07$ times the measured mean JCT of TP2.

G.2 One-Anchor Multiplicative Calibration

For simulator s and deployment d, set $\lambda_0 = 0.02$ programs/s and compute

$$\alpha_{s,d} = \frac{J_d(\lambda_0)}{\widehat{J}_{s,d}(\lambda_0)}, \qquad \widetilde{J}_{s,d}(\lambda) = \alpha_{s,d}\widehat{J}_{s,d}(\lambda). \tag{5}$$

Simulator	Raw mean error (%)	Corrected mean error (%)
SplitwiseSim	105.95	35.10
DynoSim	11.25	5.65
APEX	289.60	53.75
TokenSim	43.63	22.81
LLMServingSim 2.0	7.28	3.40
Frontier	33.49	17.50

Table 24: Post-hoc scalar calibration at the lowest rate, scored on the other four rates in each of eight deployments. The anchor settings are excluded from both columns.

Deployment fields (JSON fragment)

```
{"hardware": {
  "model": "meta-llama/Llama-3.1-70B", "gpu": "B200",
  "tp": 1, "num_instances": 2, "num_gpus": 2,
  "kv_pool_tokens": 83312, "block_size": 16}}
```

One program (JSONL, expanded for readability)

```
{"session_id": "p0", "arrival_time_ns": 1000000000,
 "sub_requests": [
  {"input_toks": 1024, "output_toks": 64,
   "tool_duration_ns": 50000000},
  {"input_toks": 1200, "output_toks": 32,
   "tool_duration_ns": 0}]}
```

Figure 6: The B200 two-instance deployment from the experiment matrix and an illustrative two-call program, using Arena's field names. Other deployment fields, policy fields, and token arrays are omitted. KV capacity is per instance.

The correction uses one real JCT measurement per deployment, eight in total. We score only $\lambda \in \{0.04, 0.06, 0.08, 0.10\}$, giving 32 equally weighted settings per simulator. Table 24 compares raw and corrected error over exactly those same settings. These raw averages therefore differ from the 40-setting averages in Table 3.

Multiplication by $\alpha_{s,d}$ preserves relative JCT growth across rates, testing response shape after removing a baseline scale difference. This post-hoc diagnostic rescales final program JCT, including tool delays, without rerunning scheduling, caching, or dependency feedback.

H Experiment Specification Example

Figure 6 shows selected deployment fields and one workload record. This program arrives at one second, issues a model call, waits 50 milliseconds after that call finishes, and issues its second call. Token counts describe the replayed computation; records can also carry token content for real execution.