AGENTSERVESIM: SIMULATION-DRIVEN POLICY DESIGN FOR AGENT SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Serving large language model (LLM) agents requires coordinating cache management, routing, and scheduling across model calls separated by tool execution. Simulating these policies is difficult because their decisions change subsequent call arrivals, retained context can be shared across programs, and cache reuse depends on both placement and resource availability. AgentServeSim addresses these coupled requirements through a program-level execution model that preserves history and cache state across LLM calls and enforces policy actions within common resource constraints. It derives arrivals from simulated completions, accounts for shared prefixes without duplicating their memory cost, and checks cache availability at admission. Across 20 comparisons with real serving, AgentServeSim predicts mean program job completion time (JCT) with 4.74% mean absolute percentage error. It also identifies the same winning policy as real serving among six candidates in all three evaluated settings. Using simulation to evaluate LLM-generated KV-management and scheduling code, we discover a policy that achieves 1.64–$2.04\times$ speedup in real serving over a default baseline using first-come-first-served (FCFS) scheduling and least-recently-used (LRU) KV cache eviction, and 1.25–$1.45\times$ over the strongest evaluated alternative in each setting.

1 INTRODUCTION

Large language model (LLM) agents execute programs that alternate model calls with external tool use (Yao et al., 2023; Jimenez et al., 2024; Patil et al., 2025). A coding agent, for example, edits code, runs a test, and then issues its next model call with the test result appended to its prior context. Serving such programs efficiently requires three coordinated decisions. While a program waits for a tool, the system must decide whether to retain, offload, or discard its key–value (KV) cache: retaining it avoids recomputation but leaves less memory for other programs (Abhyankar et al., 2024; Li et al., 2026). When the next call becomes ready, the system must choose a serving instance (Srivatsa et al., 2024; Wang et al., 2026) and decide when to run the call (Luo et al., 2025). A less busy instance may lack the cached context, and a delayed call postpones the rest of the program. The benefit of these *program-aware serving policies* therefore depends on how they are combined, and must be measured by program job completion time (JCT), the time to complete the whole program.

Simulation offers a practical way to evaluate serving decisions without replaying every candidate on hardware. Vidur, for example, explores deployment configurations for LLaMA2-70B in about one hour on a CPU machine, compared with an estimated 42K GPU-hours on real hardware (Agrawal et al., 2024). Existing simulators model parallelism, disaggregation, and prefix caching; some also support dependent arrivals, stateful requests, or cache-aware routing (Cho et al., 2026; Feng et al., 2026; NVIDIA, 2026). These capabilities capture parts of agent execution (Table 1). Evaluating interchangeable program-aware policies together additionally requires their decisions to operate on consistent program and cache state throughout execution. Adding a tool delay between requests alone does not provide this model.

A policy changes future arrivals. Delaying one model call delays its tool execution and subsequent calls. Replaying fixed call timestamps therefore misses part of the consequence of changing a policy. The simulator must release each successor from the preceding call's simulated completion and tool wait, and preserve the program history needed by later decisions. Future trace information used to generate release events must remain separate from observations available to a policy.

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.

Table 1: Comparison of LLM serving simulators on agent-serving capabilities. **Unit**: agent program as the simulation unit, with identity and history kept across turns. **Arr.**: successor turn arrival determined by simulated completion and tool duration. **KV**: program-aware retain/offload/discard of KV state across tool calls. **RT**: cache- and load-aware routing across instances. **SC**: scheduling that uses program execution history. ✓: supported, △: limited or partial, ×: not supported.

	Program state		**Program-aware policies**		
Simulator	Unit	Arr.	KV	RT	SC
Vidur (Agrawal et al., 2024)	×	×	×	△	×
LLMServingSim 2.0 (Cho et al., 2026)	×	✓	△	△	×
Frontier (Feng et al., 2026)	△	✓	△	△	△
DynoSim (NVIDIA, 2026)	△	△	△	✓	×
AgentServeSim (ours)	✓	✓	✓	✓	✓

Cache state persists and is shared. Context can remain cached during tool waits, share prefixes with other programs, or be in transit between memory tiers. Counting cached context independently for each program would overestimate memory use; treating an in-transit copy as available would underestimate recomputation or transfer delay. The simulator must track shared ownership, distinguish residency from protection, and make transferred context reusable only when the copy completes.

The decisions affect one another. Retention consumes memory, routing determines whether retained context is accessible, and scheduling can change whether that context survives until use. A destination selected for its cached context may lose that context while the call waits in its queue. These policies therefore need consistent shared state and resource enforcement, including cache and capacity checks when a call is admitted.

We present AgentServeSim, which maintains program history and persistent cache state across calls on top of a simulation backend. Its shared execution model propagates changes in retention, routing, and scheduling to later calls while respecting resource constraints. Completion-driven call release captures timing dependencies, and separation of future trace data from completed observations prevents policies from using tool information prematurely. Shared-prefix accounting charges memory once while maintaining distinct residency and protection state; admission-time checks ensure that anticipated cache reuse is still feasible. The inherited backend supplies execution timing; these additions determine program progress and policy-dependent work.

Across 20 comparisons with real serving, AgentServeSim predicts mean program JCT with 4.74% mean absolute percentage error. Accurate averages alone do not establish reliable policy comparisons: simulation also identifies the same winning policy among six candidates in three ranking comparison settings. An LLM-driven search over KV-management and scheduling code completes in 15 h 38 min on six CPU cores, compared with an estimated 279 GPU-hours for hardware evaluation. The resulting policy achieves 1.64–$2.04\times$ real speedup over a default FCFS/LRU baseline and 1.25–$1.45\times$ over the strongest alternative in each setting.

This paper makes the following contributions:

- A shared program-level execution model that connects persistent history and cache state to resource-constrained policy execution. Completion-driven arrivals, separation of future trace data from policy observations, shared-prefix accounting, and admission-time checks address fixed-arrival replay, premature information access, duplicate memory charges, and stale assumptions about cache reuse.
- A simulator-guided policy-synthesis workflow that makes repeated evaluation of executable decision rules practical. The simulator supports KV management, routing, and scheduling; the demonstrated search changes KV-management and scheduling code under a fixed deployment.
- An evaluation of prediction fidelity, policy-ranking agreement, and real-serving improvements across workloads and deployments.

2 Related Work

LLM serving simulators. Existing simulators provide execution models for studying model deployment configurations and serving optimizations. Vidur (Agrawal et al., 2024) combines profiling with simulation to evaluate inference configurations, while APEX (Lin et al., 2025) searches parallel execution plans. LLMServingSim and its 2.0 release model heterogeneous accelerators, multi-tier memory, and disaggregated serving (Cho et al., 2024; 2026). Support for dependent arrivals in LLMServingSim 2.0, stateful requests in Frontier (Feng et al., 2026), and cache-aware routing in DynoSim (NVIDIA, 2026) already captures aspects of execution across calls. The distinction in AgentServeSim is the shared execution model for evaluating interchangeable program-aware policies together (Table 1). Program history and persistent cache state connect policy decisions to subsequent arrivals, while shared-prefix accounting and admission-time checks enforce their memory and reuse consequences. This coordination layer operates above a simulation backend and makes the interactions among retention, routing, and scheduling explicit.

Agent-serving systems. Agent-serving systems establish why program dependencies and persistent context matter to serving decisions. TraceLab (Zhu et al., 2026) characterizes coding-agent workloads with long execution chains, variable tool durations, and substantial prefix reuse. Parrot (Lin et al., 2024) exposes application dependencies, Autellix (Luo et al., 2025) schedules calls using program-level attained service, and Conveyor (Xu et al., 2024) overlaps partial tool execution with decoding. For context reuse, SGLang's RadixAttention (Zheng et al., 2024) shares KV prefixes across generation calls; CachedAttention (Gao et al., 2024), Mooncake (Qin et al., 2025), and LMCache (Liu et al., 2025) provide mechanisms for multi-turn caching, storage, and reuse. InferCept (Abhyankar et al., 2024) manages preservation, offloading, and recomputation during tool interruptions, while Continuum (Li et al., 2026) combines tool-aware retention with program-level scheduling. Across instances, Preble (Srivatsa et al., 2024) balances prefix reuse and load, and SAGA (Guo et al., 2026) coordinates workflow-aware KV management, session affinity, work stealing, and fairness. AgentServeSim complements these designs by evaluating interchangeable policies over shared program and cache state, propagating their interactions to subsequent execution.

Simulation- and LLM-driven policy search. Policy-code synthesis changes the decision rules being evaluated, whereas configuration search selects among deployment choices and existing rules. Vidur-Search (Agrawal et al., 2024) explores parallelism, implemented scheduling policies, and scheduler parameters; DynoSim (NVIDIA, 2026) explores deployment and routing configurations. FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025) instead use evaluators to guide iterative generation of candidate programs. We apply this approach to agent-serving policies: an LLM proposes KV-management and scheduling code, and AgentServeSim evaluates its effect on program completion time under a fixed deployment. The policy interfaces (Table 11) allow candidates to change how program history and memory pressure determine serving actions, while the execution model enforces common resource constraints. Although the simulator also supports routing policies, the demonstrated search changes KV management and scheduling. We evaluate both simulator agreement on policy selection and the discovered policy's performance in real serving.

3 AgentServeSim Design

3.1 System Overview

AgentServeSim connects program progress and persistent cache state so that serving decisions change execution timing and subsequent calls (Figure 1b). Shared state preserves program identity, completed-call service, observed tool behavior, placement history, and associations with cached context. Above a simulation backend, the Program Orchestrator tracks progress and releases ready calls, the Global Scheduler routes and schedules them, and the KV Lifecycle Manager tracks residency and protection.

Figure 1a shows programs A and B sharing prefix P on I1. After A1 completes, its context remains protected during the tool wait; P is stored once despite both programs referencing it (1). A2 arrives after A1's simulated completion and tool duration. Routing selects I1 using locality and load (2),

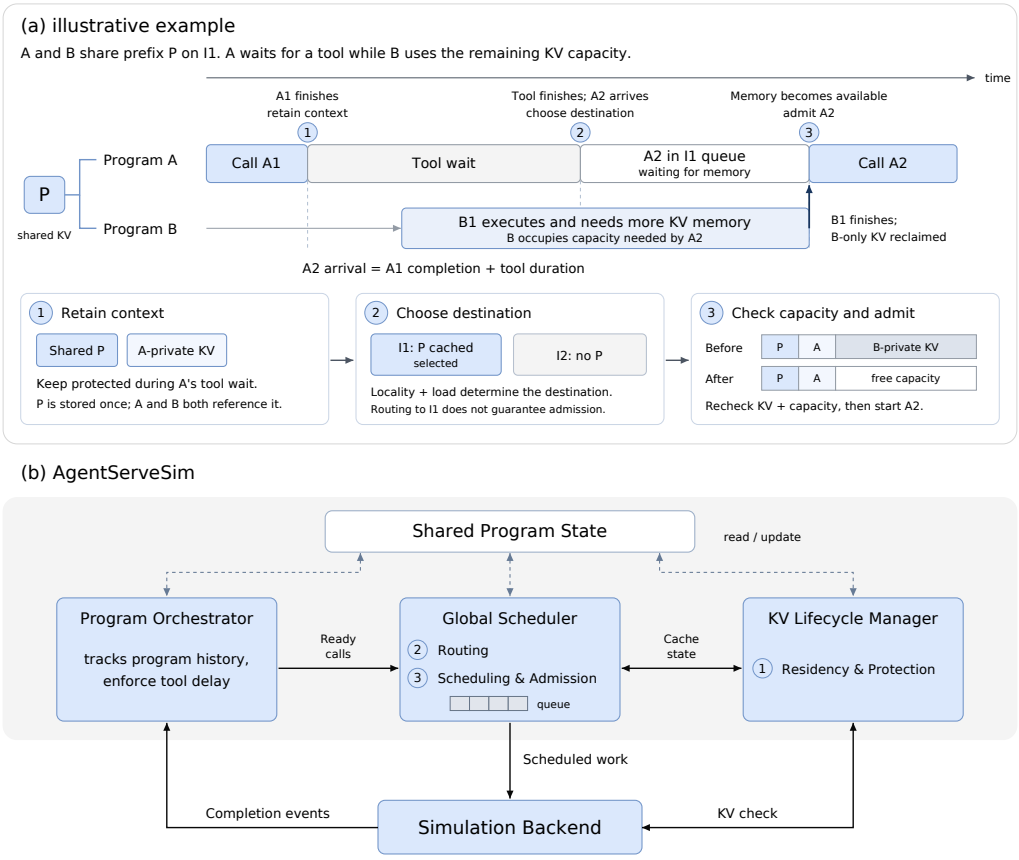

Figure 1: (a) Illustrative example linking retention (1), routing (2), and admission (3): shared context remains cached, but A2 waits for capacity despite returning to I1. (b) AgentServeSim coordinates these decisions through shared program state and execution feedback from the simulation backend.

but B1 occupies capacity needed by A2. A2 waits in I1's queue until B1 finishes and B-only KV is reclaimed, then executes after cache and capacity checks (3). Arrival, destination, and admission thus depend on distinct consequences of earlier decisions.

The simulation takes captured program calls and tool durations, a deployment configuration, hardware profiles, and serving policies as inputs. At an initial arrival or the end of a tool wait, the Program Orchestrator releases a ready call. The Global Scheduler selects its destination and places it in that instance's request queue. To form a batch, it selects calls and consults the KV Lifecycle Manager for reusable context and available capacity, accounting for any required recomputation within token, sequence, and memory limits. The scheduled batch is passed to the simulation backend, which uses profiled execution costs to return timing information. A vLLM-based layerwise profiler supplies the underlying CUDA kernel timings. Returned timing advances call progress and cache state; completion events update the program's service history and trigger retention decisions. If another call follows, the orchestrator releases it after the recorded tool wait, exposing the completed tool observation to policies at that point. This feedback loop continues until all programs finish. The policy interface in Appendix G allows decision rules to be replaced or synthesized while retaining the same state transitions and resource checks.

3.2 Program Orchestrator

A policy-induced delay must propagate to dependent calls, while accumulated history remains available to later decisions. The Program Orchestrator preserves program identity, completed-call history, attained service, and observed tool durations. Only the initial arrival is fixed; successors are released from their predecessor's simulated completion and recorded tool duration. The same captured work can therefore produce different timelines under different policies. Trace information

and policy observations remain separate: a recorded tool duration determines its release event but becomes visible through the standard policy interface only after the tool finishes. Policies can use completed observations without receiving future trace information. Program JCT spans initial arrival to completion of the final model call.

3.3 Global Scheduler

Routing and scheduling must agree on the state that makes a destination useful. Random or round-robin routing treats returning calls independently of program placement, potentially losing context that remains cached at a previous instance. Placement history alone is also insufficient: context may have been evicted, or queueing at that instance may outweigh the benefit of reuse. The Global Scheduler exposes placement history, current cache residency, and instance load to the routing policy and places each ready call in the selected instance's request queue. The destination's KV Lifecycle Manager checks actual reuse when the call is admitted. This separates a routing decision based on current locality from the later capacity and cache checks needed to execute it, accounting for evictions while the call waits.

Scheduling must preserve service history across calls while enforcing the resource consequences of its ordering. Request-scoped priorities omit service received by earlier calls, even when that service is relevant to fairness between programs. The Global Scheduler exposes accumulated program service when a policy assigns priorities, admits or defers calls, or chooses preemption victims. For least-attained-service scheduling, each completed call contributes the time from first scheduling to its last token, excluding its initial queue wait but including subsequent stalls. The chosen ordering is applied through continuous batching, with the KV Lifecycle Manager checking reusable context and capacity before prefill or decode work is admitted. A memory-preempted call returns to the queue; readmission checks what context remains and recomputes missing KV. A priority change can therefore alter both the order of execution and the amount of work required, and those changes propagate to subsequent program arrivals.

3.4 KV Lifecycle Manager

Context can outlive a model call, and several programs can depend on the same cached prefix. Prefix caching provides reuse, but residency alone does not express whether context should remain protected during a tool wait. The KV Lifecycle Manager therefore separates physical residency from the references that protect context. A radix cache adapted from SGLang's RadixAttention (Zheng et al., 2024) represents shared prefixes, charging each resident prefix once even when multiple programs use it. Active references protect context needed by execution or pending copies; retention references protect context across calls until a policy deadline or release event, subject to permitted pressure reclamation. Releasing one program's reference does not remove protection required by another. Capacity reservations separately account for space needed by scheduled work or copies (Appendix G, Table 12). This distinction lets policies change protection without duplicating memory charges or equating the end of protection with immediate eviction.

Cache reuse becomes valid only when the required state is resident and ready. Before allocating space, the manager checks whether free capacity and eligible reclamation can satisfy the request. It reclaims unreferenced state first and releases retention protection where permitted; an infeasible allocation leaves that protection intact. Newly computed prefixes become reusable at block-aligned boundaries after computation completes. For CPU offloading, a pending copy reserves destination capacity and protects its source until completion, when the destination copy becomes available. These checks prevent policies from receiving the benefit of a transfer before it finishes or retaining context without accounting for its capacity cost. They also connect retention to routing and scheduling: a policy can request reuse, but admission depends on the actual cache and resource state at that time.

4 Experimental Setup

Implementation. AgentServeSim uses LLMServingSim 2.0 (Cho et al., 2026) as its simulator backend, with ASTRA-Sim for execution timing (Won et al., 2023). We use vLLM v0.19.0 (Kwon et al., 2023) for layerwise profiling and hardware replay, with replay support for program-aware

serving policies. Simulation and hardware replay execute the same captured workloads under corresponding policy configurations.

Models and environment. Our evaluation covers Llama-3.1-8B, Llama-3.1-70B, and Phi-3.5-MoE on NVIDIA B200 and RTX PRO 6000 GPUs. These configurations span dense and mixture-of-experts models, different memory constraints, and single- and multiple-instance deployments. Each simulation uses the corresponding hardware replay's deployment configuration, KV capacity, and serving limits. Together with the matched policy configurations and captured call dependencies, this keeps the resource constraints and workload consistent across both systems.

Workloads. We use coding-agent traces from SWE-bench (Jimenez et al., 2024), including a leaderboard-derived workload denoted SWE-LB, and tool-calling traces from BFCL (Patil et al., 2025). The validation workloads contain 50 SWE-bench programs, 100 SWE-LB programs, and 150 BFCL programs. Initial program arrivals follow a Poisson process. RQ1 uses rates of $\{0.02, 0.04, 0.06, 0.08, 0.10\}$ programs/s, except Continuum, which uses $\{0.1, 0.2, 0.4, 0.8, 1.2\}$ programs/s. Each program preserves its captured sequence of model calls and tool waits. Both systems release subsequent calls from their own completion events and the recorded tool durations, allowing policy decisions to affect the rest of the program's execution. Appendix A reports trace statistics and arrival generation.

Serving policies. We study policies that use program information to guide scheduling, KV management, and routing. Autellix (Luo et al., 2025) prioritizes calls using their programs' attained service, allowing programs with less accumulated service to progress. InferCept (Abhyankar et al., 2024) chooses among preserving, offloading, and discarding context by comparing their GPU-memory waste during interruptions. Continuum (Li et al., 2026) combines tool-aware KV time-to-live decisions with scheduling that prioritizes returning calls whose context remains protected. SAGA (Guo et al., 2026) uses workflow information for cache retention and session affinity for placement. Policy comparisons also include a default baseline, denoted Stock, with request-level first-come-first-served (FCFS) scheduling and least-recently-used (LRU) KV cache eviction.

Evaluation metrics. We measure prediction fidelity using mean program JCT and its p90, p95, and p99 values. For each statistic, absolute percentage error is measured relative to real serving; MAPE averages these errors across the evaluated settings and arrival rates. We assess policy selection by ranking candidates by mean program JCT in each system and reporting Kendall's τ_b and agreement on the winning policy. Real-serving speedup measures the improvement over the stated baseline. Speedup prediction error is the absolute percentage error of simulated speedup relative to real speedup, with each system using its own Stock baseline. Appendix E also reports turn-level latency and throughput. Search configuration and evaluation settings are described in Section 5.2 and Appendix B.

5 RESULTS

5.1 RQ1: HOW ACCURATELY DOES AGENTSERVESIM REPRODUCE REAL SERVING PERFORMANCE?

Mean program JCT is our primary measure of AgentServeSim's prediction fidelity. JCT is the user-perceived latency of an agent program, measured from its initial arrival to the completion of its final model call, and the objective that program-aware policies optimize (Luo et al., 2025; Li et al., 2026). These policies often slow down one call to speed up the whole program, which per-call metrics such as TTFT and TPOT do not capture. JCT includes queueing, recomputation, and tool waits across calls, testing the accumulated effects of serving decisions on program execution. We therefore measure accuracy by the mean absolute percentage error (MAPE) in mean program JCT.

We validate AgentServeSim on four prior program-aware policies (Section 4) that together exercise KV management, scheduling, and routing: InferCept and Continuum on B200, and Autellix and SAGA on RTX PRO 6000. Together the settings cover dense and MoE models, tensor parallelism, host-memory offloading, and multi-instance serving. Each setting uses the five-rate sweep and

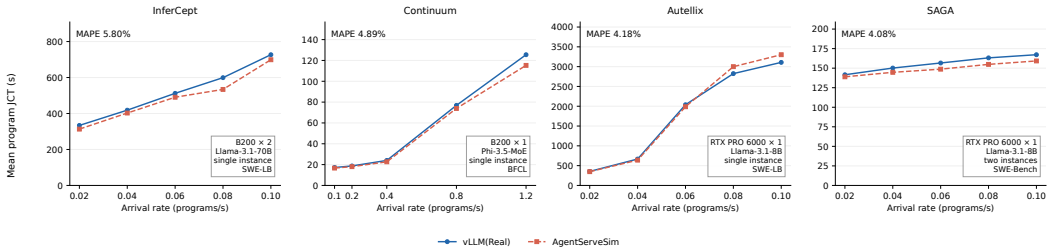

Figure 2: Mean program JCT in vLLM and AgentServeSim across four serving settings and five arrival rates. Panel titles name the deployed serving policy; deployment and workload details appear within each panel. Annotations give MAPE, the absolute percentage error relative to vLLM averaged over the five arrival rates.

completion-driven replay described in Section 4. Across the 20 comparisons, AgentServeSim predicts mean program JCT with 4.74% MAPE (Table 2).

AgentServeSim captures the increase in program JCT as load rises (Figure 2). The largest single error is 10.90%, for InferCept at 0.08 programs/s. Autellix shows the sharpest transition: real mean JCT rises from 666.8 to 2,037.8 s between 0.04 and 0.06 programs/s. Simulation places this transition at the same rate, predicting 638.9 and 1,987.1 s. Continuum's JCT grows 7.3× between 0.1 and 1.2 programs/s, and simulation stays within 8.15% throughout. SAGA stays lightly loaded across its sweep, with real JCT between 141.6 and 167.2 s, and simulation is within 5.10% at every rate.

Table 2: Mean and tail JCT MAPE over five arrival rates per policy; Overall averages all 20 comparisons.

Policy	Mean	p90	p95	p99
InferCept	5.80%	6.96%	5.63%	8.54%
Continuum	4.89%	6.27%	4.66%	4.54%
Autellix	4.18%	4.16%	4.74%	3.64%
SAGA	4.08%	6.09%	4.53%	5.02%
Overall	4.74%	5.87%	4.89%	5.44%

Accuracy carries over to the tail of the program-JCT distribution. Averaged over the 20 comparisons, p90, p95, and p99 errors are 5.87%, 4.89%, and 5.44%, respectively (Table 2). Per-setting tail errors range from 3.64% to 8.54%, close to the corresponding mean-JCT errors. Turn-level TTFT, TPOT, and throughput comparisons appear in Appendix E.

Appendix D compares these predictions with LLMServingSim 2.0 running its default policies. That comparison retains dependent call arrivals but does not reproduce the deployed program-aware policy choices.

5.2 RQ2: Can simulator-guided policy search improve real serving performance?

The large agent-serving policy space makes LLM-driven evolutionary search useful for exploring new decision rules (Romera-Paredes et al., 2024; Novikov et al., 2025), but evaluating every proposal on hardware is costly and time-consuming. We automate this loop with AgentServeSim: an LLM proposes policy code, simulation evaluates it, and the results guide subsequent proposals. We ask whether the discovered policy improves real serving, including settings unseen during search.

We use OpenEvolve (Sharma, 2025) to synthesize KV-management and scheduling code under a fixed deployment and execution model. The search starts from Seed, which protects KV context for two seconds after each call and prioritizes programs with protected context, followed by program arrival order. Candidates are evaluated on a 20-program SWE-LB subset served by one simulated Llama-3.1-70B TP2 instance with two RTX PRO 6000 GPUs. The objective is mean program JCT averaged across arrival rates of 0.05 and 0.10 programs/s. Over 50 evolutionary iterations, cached results for repeated candidates reduce the search to 41 unique policy evaluations, each covering both rates. The joint search takes 15 h 38 min on six CPU cores with 40 GB of memory and no GPUs for simulation (Table 3). Evaluating the same candidates at both rates on hardware is estimated to require 279 GPU-hours with the same cache reuse. Appendix B gives the search configuration, trajectory, and cost accounting; Appendix C gives Seed and the resulting policy.

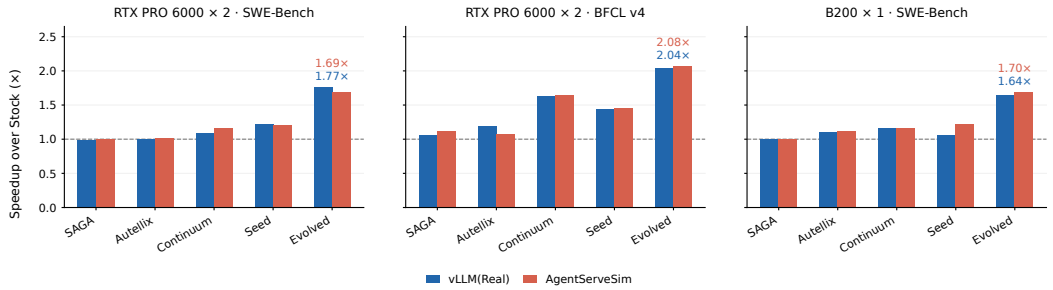

Figure 3: Speedup in mean program JCT over Stock, computed separately within vLLM and AgentServeSim. All settings serve Llama-3.1-70B on one instance. Both RTX PRO 6000 settings use TP2 at 0.10 programs/s; B200 uses TP1 at 0.06 programs/s. Search used SWE-LB on RTX PRO 6000 (TP2) at 0.05 and 0.10 programs/s; none of the plotted workload–deployment settings was used during search. Labels give Evolved's simulated and real-serving speedups.

We evaluate transfer on a single Llama-3.1-70B instance in three settings. RTX SWE-Bench, closest to the search distribution, retains the hardware, TP2 configuration, and 0.10-programs/s arrival rate but replaces the 20-program SWE-LB subset with 50 SWE-Bench programs. RTX BFCL v4 changes only the workload to 150 tool-calling programs. B200 SWE-Bench retains the 50 coding programs but changes hardware to B200, TP to one, and arrival rate to 0.06 programs/s.

Table 3: Search cost for 41 unique policies, each evaluated at two arrival rates. GPU costs are estimated.

	Simulation search	GPU replay (per instance)
CPU cores	6	12
Host memory	40 GB	96 GB
GPUs	0	2 RTX PRO 6000
Time per evaluation	62 min	3.4 h
Search wall time	15 h 38 min	—
Search GPU-hours	0	≈279

The discovered policy, Evolved, achieves the lowest real mean program JCT among the six candidates: Stock, Autellix, Continuum, SAGA, Seed, and Evolved (Figure 3). Against Stock, the default FCFS/LRU baseline, its speedups are $1.77\times$ on RTX SWE-Bench, $2.04\times$ on RTX BFCL v4, and $1.64\times$ on B200 SWE-Bench. Against the strongest evaluated alternative in each setting—Seed on RTX SWE-Bench and Continuum in the other two—the corresponding speedups are $1.45\times$, $1.25\times$, and $1.42\times$. The improvements therefore carry over to real serving beyond the workload–deployment settings used to guide the search.

5.3 RQ3: Does AgentServeSim preserve policy rankings?

Simulation must support reliable policy selection, yet small prediction errors can reverse closely matched policies even when average JCT error is low. We ask whether AgentServeSim preserves real-serving rankings, selects the same winner, and predicts that policy's improvement over Stock.

AgentServeSim identifies the same winning policy as real serving in all three settings. Ranking the six candidates from Section 5.2 by mean program JCT places Evolved first in both systems. Simulation preserves 14 of the 15 policy-pair orderings in each RTX setting and 13 of 15 on B200, giving Kendall's τ_b values of 0.867, 0.867, and 0.733 (Table 4). This agreement supports policy selection among the evaluated candidates.

Table 4: Ranking agreement across six policies and absolute percentage error in Evolved's Stock-relative speedup.

Setting	Kendall's τ_b	Speedup prediction error
RTX / SWE-Bench	0.867	4.37%
RTX / BFCL v4	0.867	1.77%
B200 / SWE-Bench	0.733	3.06%

The remaining ranking differences involve lower-ranked candidates. On RTX SWE-Bench, simulation reverses Stock and SAGA, whose simulated JCTs differ by less than 0.1%. On RTX BFCL v4, it reverses Autellix and SAGA. On B200, it places Seed ahead of Continuum and Autellix, whereas both outperform Seed in real serving. These reversals leave the selected winner unchanged. Evolved's speedup prediction error is 4.37% on RTX SWE-Bench, 1.77% on RTX BFCL

v4, and 3.06% on B200 SWE-Bench. Appendix Table 10 reports the detailed JCT and speedup prediction errors for all six policies in each setting.

5.4 Component Analysis

KV Lifecycle Manager and program-aware scheduling. Program-aware scheduling improves prediction fidelity in both evaluated deployments, while the effect of program-aware KV management depends on the setting (Table 5).

We remove each mechanism or both from the simulator while keeping the real Continuum reference fixed. The KV Lifecycle Manager ablation disables program-aware retention while preserving prefix caching and LRU reclamation; the scheduling ablation replaces program-aware ordering with request-level FCFS. On RTX PRO 6000, the full model has the lowest error, 6.26%; removing program-aware KV management or scheduling raises it to 9.80% or 8.96%. On B200, removing program-aware scheduling raises error from 4.50% to 20.43%, whereas removing program-aware KV management lowers it to 1.60%. Program-aware KV management likely has less effect in the B200 setting because its larger GPU memory and lower arrival rate of 0.06 programs/s reduce cache pressure.

Table 5: Absolute mean-JCT prediction error on SWE-Bench with Llama-3.1-70B, using the two deployments in Figure 3.

Variant	RTX	B200
Full	6.26%	4.50%
w/o KV Lifecycle Mgr.	9.80%	1.60%
w/o prog.-aware sched.	8.96%	20.43%
w/o both	8.86%	20.93%

Table 6: Simulated routing performance on SWE-Bench with Llama-3.1-70B at 0.10 programs/s. JCT is in seconds; cache hits are token-weighted. Bold marks the best values per deployment.

	RTX PRO 6000: four TP2 instances				**B200: four TP1 instances**			
Router	Mean	p95	p99	Cache hit	Mean	p95	p99	Cache hit
Cache-aware	**712.0**	1,686.4	**2,379.7**	**77.46%**	**316.7**	**771.9**	**1,124.8**	**73.81%**
Least-loaded	725.2	**1,659.5**	2,382.4	70.36%	326.6	818.0	1,186.0	60.96%
Round-robin	1,253.3	2,977.4	3,650.9	44.02%	421.5	1,084.4	1,504.8	45.49%
Random	1,396.4	3,205.7	3,877.0	40.93%	476.4	1,202.0	1,612.4	41.80%

Cache-aware routing. Routing changes the value of retained context by determining whether a returning program can access it. We compare cache-aware, least-loaded, round-robin, and random routing in simulation while holding Continuum KV management and FCFS scheduling fixed. Both deployments serve SWE-Bench with Llama-3.1-70B at 0.10 programs/s using four instances: TP2 on RTX PRO 6000 and TP1 on B200. Cache-aware routing produces the lowest mean JCT and highest cache-hit rate on both platforms (Table 6). Its mean JCT is 712.0 seconds on RTX PRO 6000 and 316.7 seconds on B200, compared with 725.2 and 326.6 seconds for least-loaded routing. Both policies outperform round-robin and random placement, showing that load balance matters alongside cache locality in these settings. Cache-aware routing also has the lowest p99 on both platforms and the lowest p95 on B200; least-loaded routing has a slightly lower RTX p95, 1,659.5 versus 1,686.4 seconds. These comparisons show how placement changes cache reuse and program latency under fixed retention and scheduling rules.

6 Conclusion

AgentServeSim captures interactions among KV management, routing, and scheduling through completion-driven arrivals, shared-prefix accounting, and admission-time checks. It predicts mean program JCT with 4.74% MAPE across 20 real-serving comparisons and selects the same winner among six policies in all three ranking settings. Simulator-guided synthesis of KV-management and scheduling code yields 1.64–$2.04\times$ real speedup over the default FCFS/LRU baseline and 1.25–$1.45\times$ over the strongest evaluated alternative in each setting.

AI USAGE STATEMENT

We used AI tools to assist with manuscript organization and language editing, plotting code, and figure preparation. Separately, LLM-driven generation of serving-policy code is part of the research methodology described in Section 5.2 and Appendix B. The authors are responsible for the paper's claims, experimental results, citations, and final content.

REPRODUCIBILITY STATEMENT

Our anonymous code repository is available at `https://anonymous.4open.science/r/agentservesim-CAB8`. Section 4 describes the software, models, hardware, workloads, policies, and evaluation metrics. Appendices A, B, and C provide workload statistics, search settings, and policy pseudocode.

REFERENCES

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. InferCept: Efficient intercept support for augmented large language model inference, 2024. URL https://arxiv.org/abs/2402.01869.

Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S. Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for LLM inference. In *Proceedings of Machine Learning and Systems*, volume 6, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/b74a8de47d2b3c928360e0a011f48351-Abstract-Conference.html.

Jaehong Cho, Minsu Kim, Hyunmin Choi, Guseul Heo, and Jongse Park. LLMServingSim: A HW/SW co-simulation infrastructure for LLM inference serving at scale. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*, 2024. doi: 10.1109/IISWC63097.2024.00012. URL https://arxiv.org/abs/2408.05499.

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.

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/2605.21312.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 111–126. USENIX Association, 2024. URL https://www.usenix.org/conference/atc24/presentation/gao-bin-cost.

Dongxin Guo, Jikun Wu, and Siu Ming Yiu. SAGA: Workflow-atomic scheduling for AI agent inference on GPU clusters, 2026. URL https://arxiv.org/abs/2605.00528.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

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 *ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Huanchen Zhang, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn LLM agent scheduling with KV cache time-to-live, 2026. URL https://arxiv.org/abs/2511.02230v7.

Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 929–945. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/lin-chaofan.

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. Version 2, revised April 29, 2025.

Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. LMCache: An efficient KV cache layer for enterprise-scale LLM inference, 2025. URL https://arxiv.org/abs/2510.09665v2.

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/2502.13965.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

NVIDIA. DynoSim. NVIDIA Dynamo Documentation, 2026. URL https://docs.dynamo.nvidia.com/dynamo/dev/knowledge-base/concepts/simulation/overview. Accessed September 20, 2026.

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.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170. USENIX Association, 2025. URL https://www.usenix.org/conference/fast25/presentation/qin.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent. https://github.com/codelion/openevolve, 2025.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving, 2024. URL https://arxiv.org/abs/2407.00023.

Jiahao Wang, Kaizhan Lin, Kaixi Zhang, Jinbo Han, Xingda Wei, Sijie Shen, Chenguang Fang, Wenyuan Yu, Rong Chen, and Haibo Chen. Smetric: Rethink llm scheduling for serving agents with balanced session-centric scheduling, 2026. URL https://arxiv.org/abs/2607.08565.

William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023. doi: 10.1109/ISPASS57527.2023.00035. URL https://arxiv.org/abs/2303.14006.

Yechen Xu, Xinhao Kong, Tingjun Chen, and Danyang Zhuo. Conveyor: Efficient tool-aware LLM serving with tool partial execution, 2024. URL https://arxiv.org/abs/2406.00059.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

Kan Zhu, Mathew Jacob, Chenxi Ma, Yi Pan, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Tracelab: Characterizing coding agent workloads for llm serving, 2026. URL https://arxiv.org/abs/2606.30560.

Table 7: Statistics of the captured workload traces. P50, P95, and P99 denote linearly interpolated percentiles. Token counts are recorded trace lengths; tool gaps are in seconds. Each workload is counted once, independent of the arrival-rate sweep.

Metric	Statistic	SWE-Bench	SWE-LB	BFCL
Programs	Count	50	100	150
Total turns	Count	2,873	4,525	1,370
Turns per program	Mean	57.46	45.25	9.13
	P50	47.00	43.00	8.00
	P95	100.00	67.20	20.55
	P99	143.35	97.28	21.00
	Max	185.00	125.00	21.00
Context tokens per turn	Mean	13,463.2	13,546.8	10,810.9
	P50	11,115.0	12,302.0	4,486.0
	P95	33,690.4	28,522.6	41,704.3
	P99	43,982.9	41,670.3	69,728.3
	Max	52,724.0	52,362.0	96,742.0
Output tokens per turn	Mean	93.7	309.1	45.1
	P50	52.0	130.0	26.0
	P95	351.2	1,198.4	83.5
	P99	929.5	1,853.0	145.5
	Max	2,048.0	2,048.0	2,626.0
Tool gap (s)	Mean	1.114	1.188	1.492
	P50	0.200	0.157	1.181
	P95	3.667	5.023	3.472
	P99	18.544	17.883	5.718
	Max	60.068	69.770	20.186

A WORKLOADS

Dataset statistics. Table 7 summarizes the captured validation traces: 50 SWE-Bench programs, 100 leaderboard-derived SWE-LB programs, and 150 BFCL programs. A turn is one model call. Turns per program are summarized over programs; context and output lengths are summarized over all turns. Context length denotes the recorded input context before generation, including any reusable prefix, rather than only newly added tokens. Token statistics describe the standard source traces before model-specific retokenization or replay-specific context construction. Tool gaps are measured between successive calls within a program, including zero-duration gaps and excluding the terminal placeholder after the last call.

Arrival generation. Initial program arrivals follow a seeded Poisson process. At rate λ, interarrival times are exponentially distributed with mean $1/\lambda$. We keep the program contents and order fixed across rates and use the same initial arrival timestamps in simulation and hardware replay. Subsequent calls become ready after the preceding call completes and its recorded tool wait elapses; these arrivals depend on execution rather than an independent Poisson process.

B POLICY SEARCH DETAILS

Development configuration. Search evaluates policy logic under a fixed deployment. We use a 20-program SWE-LB subset with 913 turns and Llama-3.1-70B on one simulated TP2 instance comprising two RTX PRO 6000 GPUs. Each policy is evaluated at 0.05 and 0.10 programs/s, with mean program JCT averaged across the two rates. Both rates are used during development; the search changes executable policy code rather than deployment configurations.

Initial policy and search space. Seed protects KV context for two seconds after a call and releases that protection when the next call arrives. Its scheduling rule prioritizes protected context, followed by program arrival order with a bounded preference for smaller contexts. The joint search changes KV-management and scheduling decisions, including admission and preemption, through executable policy code. The resulting Evolved policy conditions admission on memory pressure and

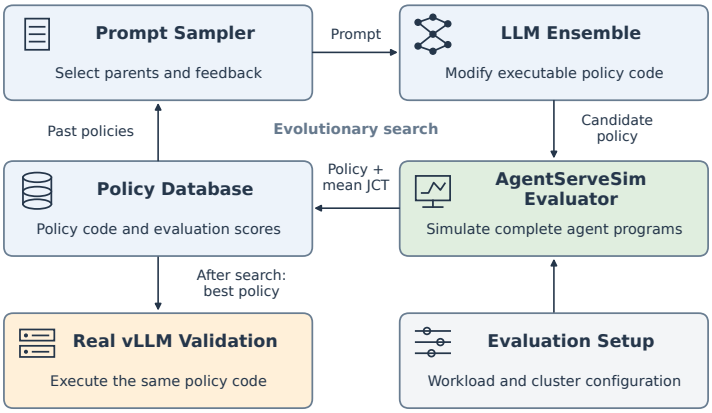

Figure 4: Simulator-guided policy search. An LLM proposes policy-code edits, and simulated performance guides subsequent proposals. Deployment configurations and the execution model remain fixed.

Table 8: Best-policy development speedup over Seed at five-iteration checkpoints, averaged across the two arrival rates. Checkpoints with identical values are grouped.

Checkpoint	Mean speedup over Seed
5, 10, 15, 20, 25, 30	1.016×
35, 40	1.302×
45, 50	1.329×

uncached context, skips protection when the moving average of observed tool gaps exceeds three seconds (1.5 times the two-second protection window), and chooses preemption victims by estimated recomputation size. The single-instance development configuration does not exercise routing choices. Appendix C gives both policies' decision rules.

Generation and evaluation. OpenEvolve proposes policy edits using an equally weighted ensemble of `anthropic/claude-sonnet-5` and `openai/gpt-5.6-sol` through OpenRouter. We run 50 evolutionary iterations with a population of 40, two islands, four concurrent evaluations, and cascade evaluation. Checkpoints are saved every five iterations. Cached results are reused when a policy has already been evaluated, yielding 41 unique policy evaluations across the two development rates.

Measured search cost. The joint search completes in 15 h 38 min using six CPU cores and 40 GB of host memory, with no GPUs allocated to simulation. Median simulation time is 30.9 minutes at 0.10 programs/s and 31.0 minutes at 0.05 programs/s, or approximately 62 minutes per two-rate evaluation on one CPU core. The search evaluates up to four policies concurrently.

Hardware-evaluation cost estimate. Hardware evaluation takes approximately 1.70 hours per policy and arrival rate on two RTX PRO 6000 GPUs, averaged over seven replays at 0.10 programs/s. Assuming the same runtime at 0.05 programs/s gives 3.40 hours per two-rate policy evaluation, or approximately 279 GPU-hours for all 41 unique policy evaluations. This estimate extrapolates from the measured arrival rate; a full hardware-based search was not run.

Search trajectory. The largest improvement appears between checkpoints 30 and 35 (Table 8), followed by a further improvement by checkpoint 45. Evolved reduces development mean JCT from Seed's 4,041.1 and 3,869.7 seconds to 3,085.4 and 2,869.2 seconds at 0.10 and 0.05 programs/s, respectively. Its mean speedup over Seed is 1.329× across the two rates. These development results are distinct from the real-serving transfer results in Section 5.2.

Transfer evaluation. We evaluate the resulting policy under the workload–deployment settings in Section 5.2. RTX SWE-Bench retains the hardware and one development arrival rate, BFCL changes the workload domain, and B200 changes the hardware, TP degree, and arrival rate. These comparisons assess transfer across operating conditions; they do not establish disjointness between all development and test programs.

C SERVING-POLICY PSEUDOCODE

Seed and Evolved share a priority rule but differ in retention, admission, and preemption. The pseudocode below describes their decision logic; the serving engine enforces allocation, protection deadlines, and execution.

Notation. Let p denote program state, t the current time in seconds, v the admission view, and C the eligible preemption candidates. The field `p.arrival` is the program's initial arrival. The flag `p.protected` denotes protection at the callback, not merely the presence of cached context. `PROTECT` requests protection until an absolute deadline; `RELEASE` removes protection while leaving context cached and eligible for eviction; `LRU` requests no additional protection or forced eviction. Smaller priority values are served first. Priorities are assigned once per released turn and do not reorder running requests. `TRUNC` truncates toward zero.

C.1 SEED

Seed requests two seconds of protection after every completed turn. It orders waiting calls by protection class, program arrival, and a bounded context-length term. Admission and preemption retain the engine defaults.

Seed policy

```
Constants: tau = 2 seconds; M = 2^40; B = 10,000
State: epoch = UNSET

ON_TURN_COMPLETE(p, t):
    return PROTECT(t + tau)

ON_TURN_ARRIVAL(p, t):
    return RELEASE

PRIORITY(p, t):
    if epoch is UNSET: epoch <- t
    a <- p.arrival if available, otherwise t
    c <- p.context if available, otherwise 0
    k <- 0 if p.protected, otherwise 1
    return k*M + TRUNC(1000*(a - epoch)) + min(TRUNC(c), B)

ADMIT(p, t, v):
    return TRUE

VICTIM(C, t):
    return ENGINE_DEFAULT
```

The context term contributes at most 10,000 priority units, equivalent to ten seconds of arrival time. Within a protection class, a shorter context can therefore precede a somewhat earlier program.

C.2 EVOLVED

Evolved retains Seed's priority rule and changes three decisions. It avoids protection when observed tool gaps suggest that the program will return well after the retention deadline, delays admissions whose uncached prompt cannot fit under high memory utilization, and selects preemption victims using prompt-plus-generated length as a recomputation-cost proxy.

Evolved policy

```
Constants: tau = 2 seconds; alpha = 0.2; threshold = 0.9
State: gap_mean = empty map keyed by tool; epoch = UNSET

ON_TURN_COMPLETE(p, t):
    estimate <- gap_mean[p.tool], if known
    if estimate is known and estimate > 1.5*tau:
        return LRU
    return PROTECT(t + tau)

ON_TURN_ARRIVAL(p, t):
    g <- completed tool-gap observation supplied to callback
    if g is available and p.tool is available:
        if p.tool is absent from gap_mean:
            gap_mean[p.tool] <- g
        else:
            gap_mean[p.tool] <- (1-alpha)*gap_mean[p.tool] + alpha*g
    return RELEASE

PRIORITY(p, t):
    return Seed.PRIORITY(p, t)  // using Evolved's own epoch

ADMIT(p, t, v):
    if p.protected:
        return TRUE
    if v.utilization is known and v.utilization > threshold:
        free <- v.free_tokens, otherwise 0
        evictable <- v.evictable_tokens, otherwise 0
        prompt <- first nonzero available value of
                  (v.prompt_tokens, p.context, 0)
        cached <- v.cached_tokens, otherwise 0
        needed <- max(0, prompt - cached)
        if needed > free + evictable:
            return FALSE
    return TRUE

VICTIM(C, t):
    if C is empty:
        return ENGINE_DEFAULT
    return first argmin over c in C of [prompt(c) + generated(c)]
```

Tool-gap estimates use only completed observations and are updated before priority assignment. The engine then releases protection and enforces capacity constraints. An admission result of `TRUE` permits an allocation attempt; it does not guarantee available capacity. The preemption rule uses token count as a proxy for recomputation cost.

D Comparison with LLMServingSim 2.0

We run stock LLMServingSim 2.0 (Cho et al., 2026) on the four validation settings of Section 5.1, using the same deployments, workloads, and arrival rates. Like AgentServeSim, it releases each successor call after the preceding call and its recorded tool wait complete. It has no interface for program-aware policies, so every run uses its defaults: FCFS scheduling, LRU prefix caching with chunked prefill, and, for the two-instance SAGA setting, round-robin routing. Errors are measured against the same vLLM runs, which deploy the named policy.

Figure 5 and Table 9 compare the two simulators. Across the same 20 comparisons, LLMServingSim 2.0 has 10.57% MAPE, compared with 4.74% for AgentServeSim. The difference depends on the setting. Under InferCept, LLMServingSim 2.0 underestimates JCT by 12.0–23.8% at 0.02–0.08 programs/s and overestimates it by 52.1% at 0.10 programs/s. For Continuum, it is within 4.0% at 0.1–0.4 programs/s but overestimates JCT by 22.9% at 0.8 programs/s. It underestimates Autellix

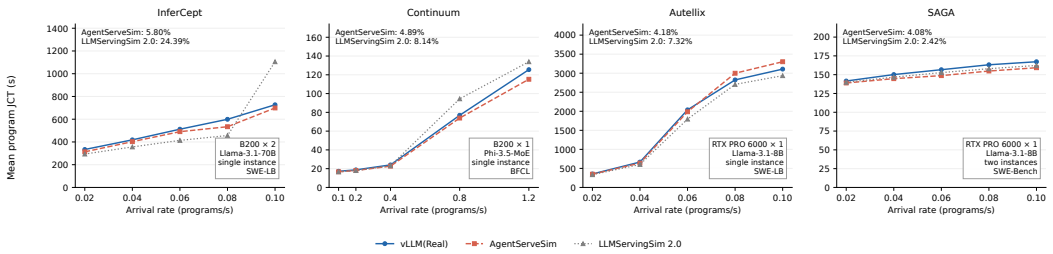

Figure 5: Mean program JCT for vLLM, AgentServeSim, and stock LLMServingSim 2.0. vLLM and AgentServeSim run the policy named in each panel title; LLMServingSim 2.0 runs its default policy. Annotations give each simulator's MAPE over the plotted rates.

Table 9: Signed error in mean program JCT relative to vLLM, $100(\mathrm{JCT}_{\mathrm{sim}}/\mathrm{JCT}_{\mathrm{real}} - 1)$, by arrival rate (programs/s). MAPE averages absolute errors over the five arrival rates.

Policy	Simulator	Signed error by arrival rate					MAPE
InferCept: B200 / TP2 / Llama-3.1-70B / SWE-LB		0.02	0.04	0.06	0.08	0.1	
	AgentServeSim	-6.2%	-3.8%	-4.2%	-10.9%	-3.8%	5.80%
	LLMServingSim 2.0	-12.0%	-14.9%	-19.2%	-23.8%	+52.1%	24.39%
Continuum: B200 / TP1 / Phi-3.5-MoE / BFCL		0.1	0.2	0.4	0.8	1.2	
	AgentServeSim	-3.3%	-3.6%	-5.6%	-3.8%	-8.2%	4.89%
	LLMServingSim 2.0	-3.4%	-4.0%	-3.7%	+22.9%	+6.7%	8.14%
Autellix: RTX / TP1 / Llama-3.1-8B / SWE-LB		0.02	0.04	0.06	0.08	0.1	
	AgentServeSim	-1.8%	-4.2%	-2.5%	+6.2%	+6.2%	4.18%
	LLMServingSim 2.0	-5.4%	-9.5%	-11.9%	-4.2%	-5.6%	7.32%
SAGA: RTX / TP1 ×2 / Llama-3.1-8B / SWE-Bench		0.02	0.04	0.06	0.08	0.1	
	AgentServeSim	-1.9%	-3.6%	-5.0%	-5.1%	-4.8%	4.08%
	LLMServingSim 2.0	-1.2%	-2.3%	-2.5%	-3.2%	-3.0%	2.42%
All comparisons: AgentServeSim (20)							4.74%
All comparisons: LLMServingSim 2.0 (20)							10.57%

JCT at every rate, by up to 11.9%. For SAGA, it is more accurate than AgentServeSim at all five rates (2.42% versus 4.08% MAPE), in this setting, where the measured JCTs under affinity and round-robin routing are close. Excluding InferCept, MAPE is 5.96% for LLMServingSim 2.0 and 4.38% for AgentServeSim.

These results show that aggregate JCT accuracy in a single setting does not, by itself, show that a simulator models the deployed policy. A simulator that runs its default policy can match JCT wherever that policy behaves like the default. It cannot distinguish policies: its prediction is the same for every policy in a comparison. Section 5.3 evaluates whether program-aware simulation preserves the ordering of distinct policies.

E Latency and Throughput Validation

Request-level latency and output throughput provide complementary views of serving behavior beyond program JCT. Figure 6 report two validation examples using Llama-3.1-8B: SAGA on two single-GPU RTX PRO 6000 instances serving SWE-Bench with 50 programs at 0.06 programs/s, and Autellix on one B200 serving SWE-LB100 at 0.04 programs/s. These examples use different deployment–workload settings and are evaluated separately.

Turn latency. Simulation reproduces the mean time to first token (TTFT) and time per output token (TPOT) within 2.73% and 6.36%, respectively (Figure 6a–b). The comparison contains 2,873 matched turns and weights each turn equally. TPOT is the interval from the first to the last output token divided by the number of output tokens minus one; single-token turns are excluded from the TPOT statistics. Real TTFT starts at engine enqueue, whereas simulated TTFT starts at simulated request arrival. Real TTFT measured from program readiness averages 68.308 ms, compared with

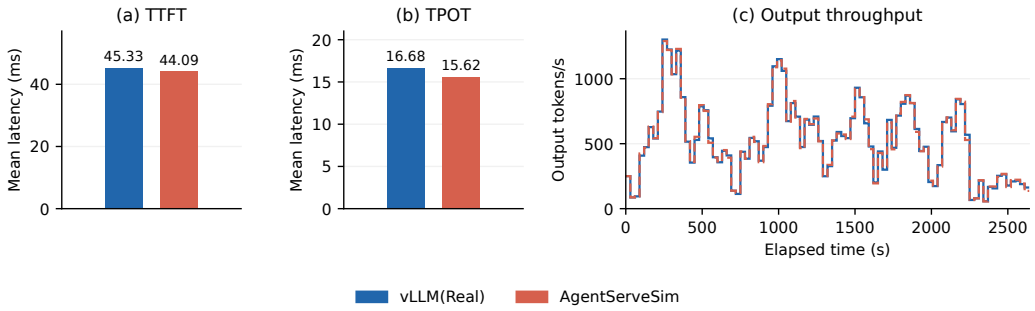

Figure 6: Latency and throughput validation against real serving. (a–b) Mean TTFT and TPOT under SAGA on two single-GPU RTX PRO 6000 instances serving SWE-Bench at 0.06 programs/s. (c) Output throughput under Autellix on one B200 serving SWE-LB at 0.04 programs/s, in 30-second bins aligned to each run's first generated-token second. Both settings use Llama-3.1-8B.

45.332 ms from engine enqueue. Thus, the engine-side comparison does not establish agreement in program-ready TTFT.

Output throughput. Simulation follows the measured throughput trajectory with 1.70% normalized absolute error and a correlation of 0.9992 at a 30-second resolution. Both runs generate 1,398,589 output tokens. Real per-second token counters and simulated token events are independently aligned to their first generated-token second, without fitting a lag or rescaling time. We retain the full duration of both traces, fill missing bins with zero, and use a 30-second denominator for every bin, including the final bin. Normalized absolute error is the sum of absolute differences in bin token counts divided by the total real output-token count. The error is 3.48% with 10-second bins and 1.10% with 60-second bins, showing the dependence of this metric on temporal resolution.

F Policy Prediction Errors

Table 10 gives the real and simulated mean program JCTs for all six policies in each RQ3 setting. All settings use Llama-3.1-70B in one instance, with TP2 on RTX PRO 6000 and TP1 on B200. Signed JCT error is $100(J_{\text{sim}}/J_{\text{real}} - 1)$; a negative value indicates underprediction. Speedup error is $100|S_{\text{sim}}/S_{\text{real}} - 1|$, where each speedup uses its own system's Stock mean JCT as the baseline. Stock therefore has zero speedup error by construction.

G Policy Interface and KV Accounting

Table 11 summarizes the observations and actions available to each policy and the execution model's enforcement of those actions. Table 12 distinguishes capacity reservations from references that protect resident context.

Table 10: Detailed policy prediction errors. JCT is in seconds; speedup errors use Stock-normalized speedups.

Policy	vLLM JCT	Simulated JCT	Signed JCT error	Speedup error
RTX PRO 6000 × 2 / SWE-Bench / 0.10 programs/s				
Stock	5,283.620	5,307.368	+0.45%	0.00%
SAGA	5,349.601	5,302.900	-0.87%	1.33%
Autellix	5,240.572	5,244.292	+0.07%	0.38%
Continuum	4,875.335	4,570.056	-6.26%	7.16%
Seed	4,346.924	4,401.930	+1.27%	0.81%
Evolved	2,991.206	3,141.845	+5.04%	4.37%
RTX PRO 6000 × 2 / BFCL v4 / 0.10 programs/s				
Stock	1,515.799	1,515.090	-0.05%	0.00%
SAGA	1,422.693	1,353.663	-4.85%	5.05%
Autellix	1,268.426	1,403.642	+10.66%	9.68%
Continuum	930.815	920.345	-1.12%	1.09%
Seed	1,046.650	1,035.894	-1.03%	0.99%
Evolved	743.160	729.925	-1.78%	1.77%
B200 × 1 / SWE-Bench / 0.06 programs/s				
Stock	1,786.341	1,865.622	+4.44%	0.00%
SAGA	1,783.912	1,860.523	+4.29%	0.14%
Autellix	1,611.813	1,665.409	+3.33%	1.08%
Continuum	1,542.717	1,612.092	+4.50%	0.06%
Seed	1,680.474	1,529.753	-8.97%	14.73%
Evolved	1,086.136	1,100.649	+1.34%	3.06%

Table 11: Policies select actions; the execution model enforces their resource consequences. Observations are examples drawn from shared program state and the views available from the engine. Unspecified decisions retain default behavior.

Policy	**Observed state**	**Decision**	**Enforcement**
KV management	Program context, completed tool history, memory utilization	Protect with a deadline, release, discard, or offload	KV Lifecycle Manager checks capacity and updates protection, residency, or transfers.
Scheduling	Program service history; queue, cache, and running-call views	Priority, admit or defer, preemption victim	Global Scheduler forms batches within token, sequence, and memory limits.
Routing	Program placement, cache residency, instance load	Destination instance	Global Scheduler dispatches the call; destination KV Lifecycle Manager checks actual reuse.

Table 12: KV accounting records. Reservations account for capacity needs; references protect resident state. These records may coexist.

Record	**Purpose**	**Release condition**
Reservation	Accounts for capacity required by scheduled work or a pending copy.	Reconciled with resident state at completion; released if the work is cancelled.
Active reference	Protects a resident prefix used by executing work or a pending copy.	Released when that use ends; other references may still protect the prefix.
Retention reference	Protects a program's cached context across calls.	Released at a policy deadline or event, or broken by permitted pressure reclamation.