MLSERVINGBENCH

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce MLServingBench, a benchmark for evaluating coding agents at a long horizon by building inference serving systems. Given a model architecture, compute topology, and workload, an agent has 12 to 36 hours in a sandbox with up to 8 H200 GPUs to build a correct, performant system. Our tasks span text generation, agentic workload, and speech and multimodal models, using production workloads. Unlike prior benchmarks that focus on localized code changes or tuning an existing engine, MLServingBench requires end-to-end system building across kernels, memory management, scheduling, and parallelism. To evaluate correctness, we introduce a token-level verifier for text model serving that compares outputs from the agent's solution against a reference implementation, enabling low-cost and high-fidelity verification. We score correct submissions by their speedup over a production serving framework. We evaluate frontier proprietary and open models across seven tasks and find that no agent outperforms the production baseline on every task, with only 23.8% of runs producing a correct system faster than the baseline. We aim for MLServingBench to guide the development of agents that can reliably build efficient serving systems for new model architectures and workloads.

1 INTRODUCTION

Coding agents have shown emerging capabilities to perform long-horizon software engineering tasks from a natural language description. Various benchmarks have been proposed to examine such capabilities in different areas ranging from software engineering (Jimenez et al., 2024; Deng et al., 2025; Miserendino et al., 2025; Chan et al., 2025; Yang et al., 2026; Zhu et al., 2026; Cognition AI, 2026; Proximal Labs, 2026), terminal interactions (Merrill et al., 2026), to LLM post-training (Rank et al., 2026) and inference optimization (Yeon et al., 2026). Yet these benchmarks differ meaningfully from *building practical production systems*. Many are constructed to produce incremental improvements to existing projects rather than end-to-end systems building, which requires extremely long-horizon reasoning (Kamahori et al., 2026). Additionally, their heavy reliance on LLM-generated tests introduces correctness and coverage risks that may be hard to audit and surface.

To more reliably measure long-horizon agentic systems building, a benchmark needs to address several gaps. First, the verifier should provide high-signal correctness feedback to detect reward hacking by agents. Moreover, scoring should reward improvements beyond known implementations despite potential contamination during training. Additionally, the tasks should be challenging and have a high ceiling for success so that no golden answers exist that could saturate the score.

One class of problems that fits such criteria is the machine learning (ML) serving stack. The serving system is a thriving research area, and the optimization space is large and open-ended. Architecting and building an ML inference stack entails multiple layers: the individual kernels (Dao et al., 2022; Ye et al., 2025), memory management (Kwon et al., 2023), intra- and inter-device parallelism (Pope et al., 2022; Zhu et al., 2025a), networking for distributed inference Patel et al. (2024); Zhong et al. (2024), scheduling (Yu et al., 2022), and the serving API (Zheng et al., 2024). Building a performant serving stack thus covers skills ranging from parallel programming to performance engineering.

Meanwhile, expertise to build such a stack is in short supply. The pace of model releases is higher than ever, increasing the demand for expertise in inference systems optimization. Existing open-source inference stacks Kwon et al. (2023); Zheng et al. (2024) can provide high performance, but

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.

focus primarily on Transformer-based models and require substantial expert effort to develop new optimizations that do not fit them Jha et al. (2026). At the same time, many model developers and researchers lack the resources needed to develop an efficient inference stack for models that deviate significantly from the Transformer architecture where infrastructure is mature, mirroring the *hardware lottery* (Hooker, 2020). As a result, exploration of new model architectures slows at a time when it is most needed to lower serving costs and expand model capabilities.

We propose a new benchmark, MLServingBench, in which a model–scaffolding pair develops an inference serving system from scratch for a specified model architecture, workload, and compute topology. The agent is given a GPU sandbox, a development toolkit, and a load emulator. We assemble a diverse set of tasks targeting generic, agentic, and multimodal serving workloads on different model architectures.

To evaluate the correctness of the agent solution, MLServingBench supports end-task metrics and provides a token-level verification scheme for tasks where end metrics are unavailable, too noisy, or too expensive to run. Agents that fail the correctness verifier receive a zero on the task. Correct solutions are then scored against a baseline that represents a production serving engine for that workload to calculate a speedup.

We found that across six frontier models and seven tasks, no model outperforms the production baseline on every task. Four of the six models fail to produce a working server in more than half of their runs; even Claude Opus 5, which has the highest overall success rate, fails in 23.8% of runs. For five of the six models, fewer than 30% of runs produce correct servers that are faster than the production baseline.

2 MLSERVINGBENCH

2.1 TASK AND BENCHMARK CONSTRUCTION

MLServingBench designs tasks around three objectives: **realistic workloads** spanning language, agentic, and multimodal serving; **ultra-long horizons**, with 12-hour builds and extensions to 36 hours; and **building from scratch**, including kernels, memory management, parallelism, batching, and scheduling.

Each task specifies a checkpoint, hardware, and workload (Table 1). Agents build in a live GPU sandbox with toolchains and profilers, then submit for evaluation in a fresh container (Figure 1). A separately containerized workload generator sends HTTP requests and records outputs and timings. Correctness and performance are assessed from the same run; failed verification yields zero.

The suite spans dense and hybrid models on one to eight H200 GPUs. For example, `agentic_coding` serves GLM-5.3-Flash across eight GPUs (GLM-5 Team, 2026; NVIDIA, 2026), requiring recurrent-state and attention-cache management alongside distributed expert computation. Tool pauses make cache retention a scheduling decision: idle sessions compete with active ones for memory (Li et al., 2025). Tasks, verifiers, and scorers are pluggable.

2.2 CORRECTNESS VERIFICATION

Generative models are inherently nondeterministic, making it difficult to guarantee or even define correctness for a serving implementation. Using end-task accuracy as the verifier is a natural way of validating that a serving system performs on par with a reference implementation. Yet in practice, end-task accuracy alone may not be sufficient to detect incorrect implementations nor fully cover the intended workload. In addition, some benchmarks can be noisy and costly to run, especially for agentic workloads. Thus, in MLServingBench, we use a combination of verifiers, including task-level where appropriate with a threshold of 98% relative accuracy compared to the reference implementation, and token-level otherwise (see Appendix D).

Meanwhile, token-level verifiers must be discriminative enough to detect system bugs while being tolerant enough to accommodate numerical differences between correct implementations. Such differences are common because floating-point arithmetic is non-associative, so even different runs of the same implementation can produce different outputs due to changes in the batch schedule.

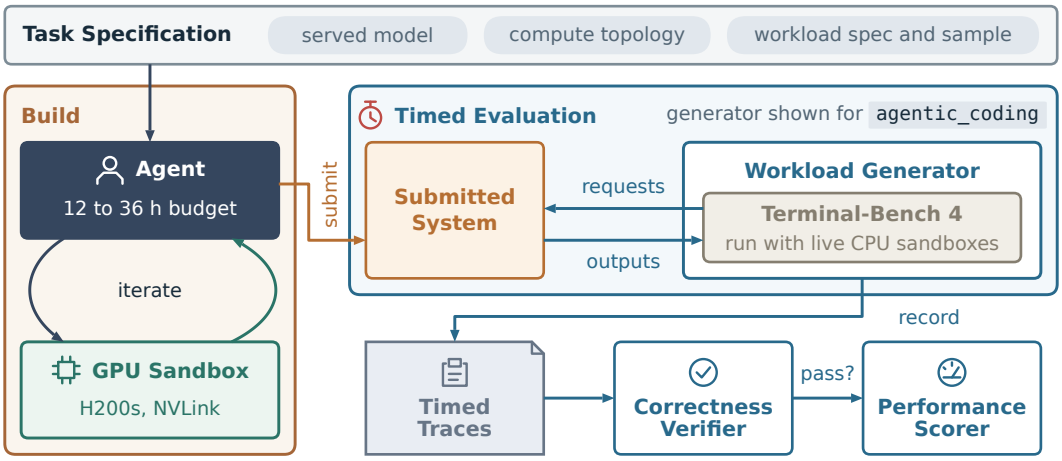


Figure 1: **The MLServingBench framework.** The final committed server is evaluated using live or trace-based workloads. The `agentic_coding` generator shown here runs held-out Terminal-Bench 4 problems in live sandboxes. Verification and scoring share the recorded trace.

Table 1: The task suite (12-hour builds). Checkpoints use bf16 except `agentic_coding` (fp8) and `speech_recognition` (fp16). Verification is token-level except for `speech_recognition` and `omni_chat`, which use task metrics; `speech_synthesis` also requires a log-mel PSNR check. Appendix D.1 gives detailed specifications and workload sources.

Task	Hardware	Description
`generic_serving` Qwen3.6-27B	1×H200	Generic API serving with multiple tenants and realistic workload patterns from the Alibaba Bailian trace (Wang et al., 2025).
`agentic_coding` GLM-5.3-Flash	8×H200	Long-horizon coding-agent trajectories on Terminal-Bench 4.0 (Merrill et al., 2026), with growing contexts and shared prefixes.
`deep_research` Gemma-4-31B-it	1×H200	Deep research agent on BrowseComp-Plus (Chen et al., 2025) with heavy retrieval prefill.
`code_completion` Mellum2-12B-A2.5B-Base	1×H200	Fill-in-the-middle completion on Real-FIM-Eval (Gong et al., 2025) typing sessions and SAFIM infills at high request rates.
`speech_recognition` Whisper-large-v3	1×H200	Audio transcription under a fixed open-loop load, scored on SLO-normalized latency.
`speech_synthesis` Qwen3-TTS-12Hz-1.7B-CustomVoice	1×H200	Streaming text-to-speech; sampling contract (§2.2.1) on codec codes.
`omni_chat` Qwen3-Omni-30B-A3B-Instruct	2×H200	Mixed image (visual QA) and speech (translation) requests on an omni model.

Each task using token-level verification uses the contract corresponding to its decoding rule. Greedy verification checks token choices; sampling verification checks both agreement with the reference distribution and consistency between reported probabilities and emitted tokens.

2.2.1 Token-Level Verifier

To evaluate decisions along the recorded trace, we teacher-force the submission's emitted tokens through a trusted reference implementation (Appendix A.1). At each position t, the reference produces a next-token distribution p_t conditioned on the prefix emitted by the submission. We calibrate

numerical tolerances using the disagreement between this reference and other trusted production serving stacks.

Greedy contract. For an emitted token y_t, define its *gap* as $g_t = \log \max_v p_t(v) - \log p_t(y_t)$. A positive gap is a *flip*, i.e. the submission emitted a token different from the reference's highest-probability token. We bound both the frequency and severity of flips. Let D be the number of flips over N scored positions, including stopping decisions, and let D_l be the number of flips with $g_t \geq c_l$ for different levels of severity $0 < c_1 < \cdots < c_L$. A submission passes when

$$\frac{D}{N} \leq \tau, \qquad D_l \leq K_l(N) \quad \text{for all } l. \tag{1}$$

The flip-rate threshold τ allows disagreements due to numerical variation near ties. The count thresholds $K_l(N)$ bound the number of larger gaps.

Sampling contract. The submission reports top-k token probabilities at each position. The goal is to check that the reported probabilities are close to the reference's and that the emitted tokens follow the reported distribution.

We first group all tokens outside the top-k into an *other* category and sum their probabilities. We apply the same grouping to the reference and submission implementations, giving distributions p_t and q_t respectively. The emitted token remains y_t; x_t denotes its grouped outcome, either y_t itself or *other* when the token is outside the retained top-k.

Agreement with the reference. We compare the mean KL divergence with a calibrated threshold κ:

$$D_{\mathrm{KL}} = \frac{1}{N} \sum_{t=1}^{N} \mathrm{KL}(q_t \,\|\, p_t) \leq \kappa. \tag{2}$$

Mean KL can miss rare premature endings, so we separately test whether the trace contains too many endings at positions where the reference assigns them low probability. Appendix A.4 defines this ending-count check.

Consistency of sampling. We apply the randomized probability integral transform (PIT) (Dunn & Smyth, 1996) to each emitted outcome. When outcomes are sampled from the reported conditional distributions, the transformed values are uniformly distributed. We assess uniformity using the Cramér–von Mises (CvM) test, also used for response-level LLM auditing by Zhu et al. (2025b). We also use a two-sided z-test on aggregate log-probability residuals to check whether emitted outcomes have the expected log probability under the reports (Appendix A.4).

We use $\alpha = 0.01$ for each of the uniformity, log-probability, and ending tests. We additionally bound the fraction of emitted tokens outside the top-k (Appendix A.4).

Calibration. We calibrate τ, $K_l(N)$, and κ using trusted engines (e.g. vLLM, SGLang, vLLM-Omni) serving the task's traffic. Calibration measures variation across these stacks and repeated runs, with upper confidence bounds for the estimated rates and means. Appendix A gives the threshold construction and complete acceptance rules. A submission that fails any required check receives a score of zero.

2.3 Performance Scoring

A submission that passes verification is scored on latency relative to each tenant's service-level objective (SLO) under the task's fixed arrival process. We use open-loop load generation so that a submission cannot be rewarded for failing to complete requests.

Tenant t declares a time-to-first-token (TTFT) budget a_t and a time-per-output-token (TPOT) budget b_t; request i of tenant t has SLO-normalized latency $r_i = \max\left(\frac{\mathrm{TTFT}_i}{a_t}, \frac{\mathrm{TPOT}_i}{b_t}\right)$.

For tasks scored by trajectory makespan, r_i is instead the trajectory's makespan divided by its SLO. We compute the geometric mean of $1/r_i$ over the N requests or trajectories and divide by the baseline value B, profiled in the same way for a production serving framework on the same workload and arrival rate. Repeated baseline runs are combined by their geometric mean during task calibration.

S is the final score and measures speedup over the baseline:

$$S = \frac{1}{B} \exp\left(\frac{1}{N} \sum_{i=1}^{N} \log \frac{1}{r_i} \right). \tag{3}$$

A given multiplicative improvement contributes equally on each request or trajectory, so applying it across the workload yields a larger gain than applying it to a subset. Requests or trajectories are weighted equally, so the score reflects the workload mix.

3 EXPERIMENTS

3.1 MODELS AND SCAFFOLDING

We select six proprietary and open models that have strong coding performance: Claude Opus 5, Claude Opus 4.8, GPT 5.6 Sol, GPT 5.5, GLM 5.3, and Kimi K3. For each model, we use the maximum available reasoning effort. Proprietary models use their official scaffolding (e.g. Claude uses Claude Code). Because open models are typically post-trained on a variety of scaffolds, we select Claude Code as the scaffolding for the open models (GLM 5.3's official evaluation uses Claude Code). We made the above model selection based on capability and availability. Since our tasks are highly challenging for the selected models, we chose to omit evaluating less capable models. We are unable to evaluate stronger models such as Claude Fable 5.1 as most tasks trigger its safety classifier immediately.

3.2 EVALUATION ENVIRONMENT AND PROTOCOL

Agents develop in containers on NVIDIA HGX H200 systems with implementation and profiling tools, read-only model checkpoints, and local development data; the scored test split is withheld. Internet access is disabled, and package installation excludes high-level inference frameworks and optimized kernel libraries (Appendix B).

Each task has a reference implementation for correctness and a performance baseline. Our infrastructure allows users to provide custom reference implementations as well. For every task, the correctness criteria are disclosed to the agent, and the agent can check its implementation against the reference during development.

Evaluation metrics. Main evaluations use three independent 12-hour builds per model and task. We enable native goal mode in both Codex and Claude Code to sustain optimization throughout this budget, analogous to the goal-mode continuation evaluated in EdgeBench (Zhu et al., 2026). We report the maximum final score for each task. Any run that fails to submit, fails to start a server, hangs, or fails the correctness gate receives a score of zero. We document the reason for each zero in the released traces. Following KernelBench (Ouyang et al., 2025), we define fast_p for $p \geq 0$ as the fraction of outcomes that pass correctness and achieve a speedup strictly greater than p. We compute this fraction over all 21 runs per model and, separately, over the seven task maxima. In the all-run aggregation, fast_0 is the success rate and fast_1 is the fraction of runs that beat the baseline.

4 RESULTS

Claude Opus 5 achieves the highest maximum score on every task and the highest overall success rate, 76.2%, followed by Opus 4.8 at 71.4% (Table 2). Opus 5 exceeds the baseline in 15 of 21 runs and on six of seven task maxima (Figure 2); no model exceeds the baseline on `agentic_coding`. We observe a larger gap between frontier open models and the strongest proprietary model than developer-reported results on other agentic coding benchmarks such as Terminal-Bench 2.1 (Merrill et al., 2026) and DeepSWE (Huang et al., 2026) suggest (Anthropic, 2026; Z.ai, 2026; Moonshot AI, 2026). We hypothesize that the complexity of our tasks and their long horizons contribute to this gap. A small mistake at any layer of the serving stack, from a data race in a GPU kernel (Appendix C.3) to an incorrect prefix-cache state (Appendix C.5), can lead to a rejection by the correctness verifier.

Figure 3 compares performance with development cost across models and across the effort and time-budget settings analyzed below.

Table 2: **Main results.** Task cells show maximum speedup over three repeats; 1 matches the baseline. fast_0 is the success rate and fast_1 is the fraction of runs that are correct and outperform the baseline, both reported as percentages of all 21 runs per model.

Model	Generic serving	Agentic coding	Deep research	Code compl.	Speech recog.	Speech synth.	Omni chat	Success rate (fast_0)	fast_1
Opus 5	1.91	0.54	1.83	1.82	25.87	2.40	4.92	76.2%	71.4%
Opus 4.8	0.45	0.029	0	0.43	7.10	1.08	0.76	71.4%	19.0%
GPT 5.6 Sol	0.012	0.013	0	0.54	0	1.66	0.54	47.6%	9.5%
GPT 5.5	0	0	0	0	0.25	0	0.005	19.0%	0.0%
Kimi K3	0.00066	0.0047	0	0	9.81	2.05	4.44	38.1%	28.6%
GLM 5.3	0	0	0	0	4.88	0	1.20	23.8%	14.3%

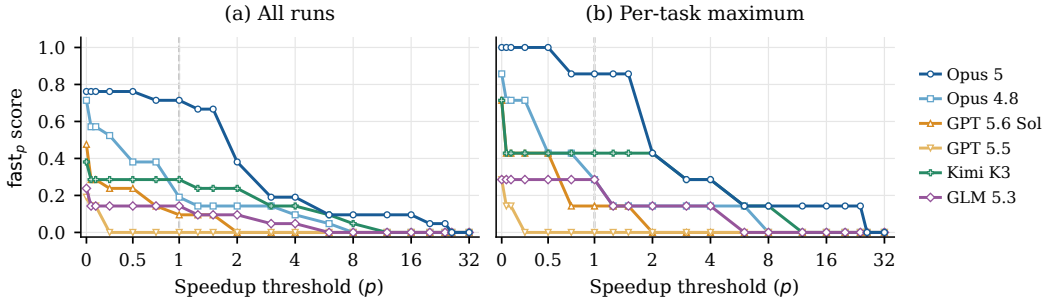


Figure 2: **Correctness and speed across all seven tasks.** fast_p is the fraction of outcomes with a passing verdict and speedup strictly greater than p. (a) fast_p across all tasks and repeats for each model. (b) The seven task maxima, each taken over 3 repeats. The horizontal scale is linear through 1 and logarithmic above 1; dashed lines mark baseline parity.

4.1 Ablations

4.1.1 Wall-Clock Budget

We run Claude Opus 5 in Claude Code at Max effort on `generic_serving` with budgets of 3, 6, 12, 24, and 36 hours, using three independent builds per budget. The served model, hardware, workload, correctness thresholds, and baseline are fixed; the 12-hour results reuse the main runs.

Longer development budgets improve the best observed serving performance (Figure 3(b)). At three hours, all three builds pass correctness but remain slower than the baseline, with a maximum score of 0.63; the six-hour maximum reaches 1.72. Additional time remains useful beyond our standard 12-hour budget: all three 24-hour builds outperform the best 12-hour build, and the maximum rises from 1.91 to 2.41 (+26%). The extension from 24 to 36 hours yields a much smaller performance gain relative to its cost: the maximum increases by 3.1% to 2.49, while total API and GPU cost across three builds rises by 29% ($2,085 to $2,690).

4.1.2 Reasoning Effort

We vary Claude Opus 5's reasoning effort across Low, Medium, High, and Max, with three independent builds per level and a 12-hour budget on `generic_serving`. Max reuses the main runs with the same task, calibration, and baseline.

Higher effort improves the best observed performance within this fixed budget (Figure 3(b)): the maximum score increases from 1.34 at Low to 1.91 at Max (+43%). This gain comes with a smaller increase in development cost: total API and GPU cost across three builds rises by 19%, from $690 to $820. Context processing dominates API spending: aggregated over the three builds at each level, cache reads and writes account for 92–94% of API cost, and output generation for 6–8%.

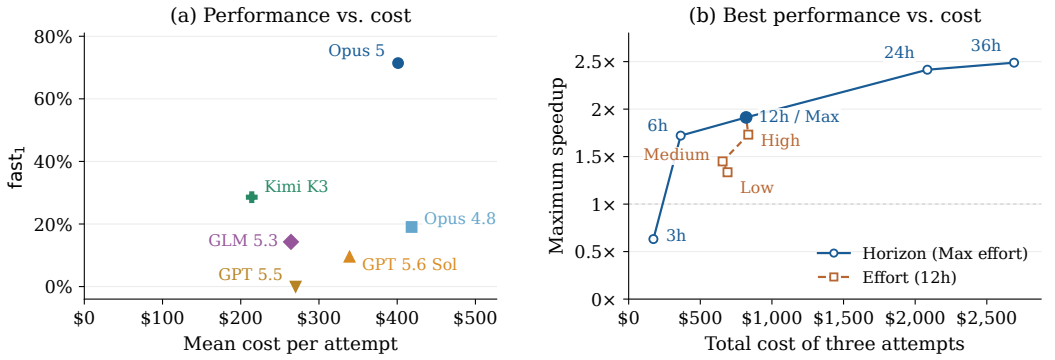


Figure 3: **Performance versus development cost.** (a) fast_1, the fraction of runs that pass correctness and beat the baseline, versus mean cost per attempt for six models across all seven tasks and three repeats per task, with 12-hour budgets and maximum reasoning effort. (b) Maximum speedup across three repeats versus their total cost for Opus 5 on `generic_serving`, varying the time budget at Max effort or reasoning effort at 12 hours. The dashed horizontal line marks baseline parity.

	CUDA graphs	Custom kernels	Paged KV†	Chunked prefill†	Prefix reuse†	Speculation†	Quantization	torch.compile
Opus 5	100%	100%	83%	75%	67%	67%	10%	0%
Kimi K3	90%	95%	50%	42%	67%	0%	29%	10%
GPT 5.6 Sol	76%	81%	25%	33%	67%	17%	19%	43%
Opus 4.8	86%	52%	33%	58%	50%	17%	0%	19%
GLM 5.3	76%	71%	25%	25%	33%	0%	0%	14%
GPT 5.5	0%	19%	0%	50%	83%	0%	0%	24%

Figure 4: **Serving optimizations in the submitted servers.** Each cell is the percentage of a model's runs whose server code includes the optimization. Optimizations marked † use the 12 runs on the four text-generation tasks; all other columns use all 21 runs. The custom-kernels column indicates that at least one kernel in the submitted server was written by the agent. On `agentic_coding`, where FP8 serving is required, quantization counts only precision below FP8. The presence of an optimization does not indicate its correctness or quality of implementation.

4.1.3 Incremental Improvement

We give Claude Opus 5 the source of vLLM 0.22.1, compiled and installed in editable mode, with Max effort and the same 12-hour budget on `generic_serving`. Across three independent repeats per condition, starting from vLLM yields a maximum score of 1.22 with one successful run, versus 1.91 with two successful runs from scratch, reusing the Max-effort main runs. All three vLLM-based servers complete the workload; the two failures are correctness rejections.

These runs highlight the difficulty of validating interactions with existing optimizations. The failed runs miss scheduler and prefix-cache defects; the passing run tests concurrent, multi-turn sessions and disables problematic optimizations (Appendix D.4).

4.2 Agent Behavior Analysis

Figure 4 reports the serving optimizations present in each model's submitted servers. Custom kernels and CUDA graphs are common: every model except GPT 5.5 ships custom kernels in at least half of its runs and CUDA graphs in at least three quarters. The models differ more in optimizations specific to text generation. Opus 5 ships a paged KV cache in 10 of its 12 text-generation runs and speculative decoding in 8, while no other model speculates in more than 2. Opus 5 is also the only

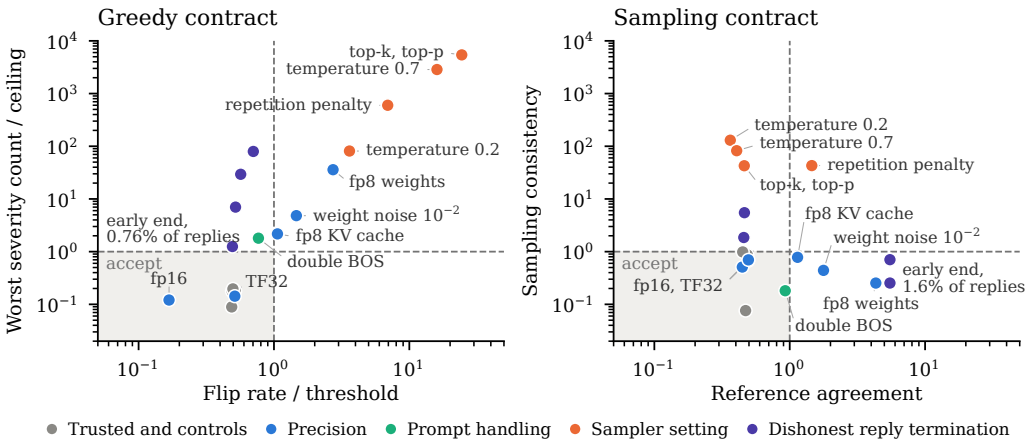


Figure 5: **Verifier decisions on controlled server variants.** Both panels show the same 18 variants; points inside the shaded square pass, and crossing either dashed boundary rejects. Each axis takes the largest normalized value among its checks (Appendix A.5). Reference agreement checks probability reports and ending behavior; sampling consistency checks emitted tokens against their reported probabilities.

model that exceeds the baseline on a text-generation task (Table 2). Kimi K3 ships custom kernels and CUDA graphs almost as often as Opus 5, but 13 of its 21 runs score zero, 8 of them because the server corrupted its cache state, ran out of memory, deadlocked or crashed while serving. Only 2 of its 12 text-generation runs pass, both far below the baseline, and none speculates; all six of its runs that beat the baseline are on the speech and omni tasks. GPT 5.5, the only model that never exceeds the baseline, ships no CUDA graphs and ships custom kernels in 4 of its 21 runs. Prefix reuse appears in 33% to 83% of every model's text-generation runs. Every custom GPU kernel in these runs is written in Triton, except for one CUDA C++ all-reduce kernel. We note that these counts record which optimizations a server contains, not how well they are implemented.

4.3 Verifier Evaluation

We evaluate 18 controlled variants of vLLM and SGLang on `generic_serving` with Qwen3.6-27B, under greedy decoding and sampling at temperature 1 (Figure 5). Both contracts accept the stock engines and tested changes to batching and sampler backend, as well as FP16 and TF32. They reject FP8 cache and weights, altered sampling settings, and forced early endings. Appendix A.5 gives the experimental details.

On six real defective servers from our rollouts, our verifier rejects all six, while InferenceBench's verifier (Yeon et al., 2026) accepts four (Table 3). Of the four incorrectly accepted servers, one triggers its defect during InferenceBench's workload; the other three require conditions that workload does not exercise (Appendix C). These cases expose two ways defects escape detection: answer accuracy can remain high despite token-level errors, and the evaluation workload can leave faulty execution paths untested.

5 Related Work

Long horizon agentic benchmarks. Agentic software-engineering benchmarks have expanded from isolated issue resolution toward sustained, repository-scale development. SWE-Bench Pro introduces larger tasks requiring substantial multi-file modifications, while SWE-EVO and Roadmap-Bench evaluate release-scale software evolution from high-level requirements (Deng et al., 2025; Le et al., 2025; Xu et al., 2026). Long-Horizon-Terminal-Bench evaluates open-ended workflows requiring hundreds of interactions and minutes to hours of execution, and PostTrainBench gives agents ten hours on an H100 to autonomously develop an LLM post-training pipeline (Li et al.,

Table 3: **Verifier comparison on real defects from our agent rollouts.** The six defects, from main and ablation runs, are detailed in Appendix C. *IB quality gate* is InferenceBench's verifier; we show the lower of two MMLU-Pro accuracies as a percentage of reference accuracy (IB sets a threshold of 95% or higher to pass). *Ours* shows a rejecting check and its statistic as a multiple of its threshold. ✓: pass; ✗: fail. The served model is Gemma-4-31B in the first row and Qwen3.6-27B otherwise.

Defect	IB quality gate	Ours
Untracked attention tensor lifetimes	✗ 19.6%	✗ mean KL, 1,470×
Slot compaction ends co-running replies	✗ 70.1%	✗ flip severity, 305×
Race in the recurrent kernel's replay	✓ 97.3%	✗ flip severity, 161×
CUDA-graph padding rows overwrite live KV	✓ 98.1%	✗ flip severity, 6.3×
Prefix-cache hit resumes a misaligned state	✓ 99.5%	✗ flip severity, 3.4×
Prefix cache with speculation skips a KV block	✓ 97.6%	✗ flip severity, 18×

2026; Rank et al., 2026). MLServingBench shares this emphasis on sustained planning, experimentation, and debugging, but focuses specifically on full-stack inference systems, where agents must optimize across different layers to deliver correct and performant stacks.

Agentic optimization of inference systems. LLM inference systems improve efficiency through coordinated optimizations across GPU kernels, KV-cache management, batching, scheduling, and parallel execution (Kwon et al., 2023; Zheng et al., 2024; Agrawal et al., 2024; Zhong et al., 2024). Recent work such as VibeServe (Kamahori et al., 2026) has begun to automate this process.

Existing performance-oriented benchmarks mostly cover isolated levels of the inference stack. KernelBench evaluates the correctness and speed of generated GPU kernels, while FlashInfer-Bench and FastKernels measure the end impact of generated kernels on serving systems performance (Ouyang et al., 2025; Xing et al., 2026; Oliaro et al., 2026). ISO-Bench evaluates agentic ability to produce patches for existing inference systems (Nangia et al., 2026). In contrast, MLServingBench evaluates end-to-end generation of a complete inference stack.

InferenceBench similarly measures whether agents could optimize LLM serving (Yeon et al., 2026) with a 2-hour time horizon. However, nearly all of its measured runs tuned vLLM arguments rather than generating and optimizing new inference systems, likely due to an insufficiently long time horizon and unrestricted Internet access. We also find that InferenceBench's verifier is unable to detect several real implementation defects (Section 4.3).

6 Discussion and Conclusion

Limitations. A potential limitation of the token-level verifier is false rejection: a correct serving implementation could exhibit numerical differences beyond the tolerances calibrated from trusted stacks. There are too few independently developed production serving engines to evaluate this risk across a diverse set of correct implementations. For every verifier-rejected submission in our evaluation, we inspected the code and identified a concrete mechanism that can explain the failure; Appendix C details six representative cases. The correctness criteria are also disclosed to agents, and every task provides reference access for self-checks before submission.

Conclusion. MLServingBench evaluates whether coding agents can build complete inference serving systems from scratch under realistic workloads. By checking the outputs of the same run used to measure performance, it makes correctness and speed joint requirements. Across seven tasks, agents produce systems that can outperform mature serving baselines, but success remains inconsistent, and no evaluated model exceeds the baseline on `agentic_coding`. The results show both the promise of autonomous systems building and the difficulty of preserving correctness while optimizing across kernels, memory management, and scheduling. We hope MLServingBench provides a useful testbed for agents that turn sustained experimentation into reliable, efficient systems.

AI USE STATEMENT

We used generative AI tools to assist with writing and polishing the manuscript and to help develop the benchmark harness and supporting infrastructure. All code produced with the help of LLMs is reviewed in pull requests by at least two authors. The authors take responsibility for the final content of this work.

ETHICS STATEMENT

MLServingBench is intended to support research on reliable and efficient inference serving. Improving serving efficiency and autonomous systems-building capabilities can broaden access to machine learning, but may also lower the cost of harmful applications. Our correctness checks assess agreement with reference implementations on the evaluated workloads; they do not establish that a generated system is secure, fair, or safe to deploy. The underlying models and existing text, code, image, and speech datasets may contain biases, sensitive information, or harmful content. Agents operate in containerized environments with restricted network access, as described in Appendix B. We document hardware allocations and build budgets there, and estimated build costs in Appendix D.3, to make the computational demands of this research explicit.

REPRODUCIBILITY STATEMENT

We will release our benchmark harness code and traces used for the main experiments. Section 2 describes the benchmark construction, correctness criteria, and scoring, while Section 3 specifies the evaluated models, scaffolding, and repeated-run protocol. Appendix A details the token-level verifier's formulation, calibration, and evaluation. Appendix B documents the hardware, software environment, data and reference access, dependency restrictions, and submission procedure. Appendix D.1 lists task-specific model checkpoints, baseline versions, decoding rules, correctness gates, workload sources, and traffic characteristics; Appendix D.2 provides an example agent build brief. Appendix C documents representative implementation defects with code excerpts to support inspection of correctness failures.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. 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*, 2024.

Anthropic. System card: Claude Opus 5. Technical report, Anthropic, July 2026. URL https://www.anthropic.com/claude-opus-5-system-card. July 24, 2026; Accessed September 20, 2026.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering, February 2025. URL http://arxiv.org/abs/2410.07095. arXiv:2410.07095 [cs.CL].

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. BrowseComp-Plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025. URL https://arxiv.org/abs/2508.06600.

Cognition AI. Introducing FrontierCode, June 2026. URL https://cognition.com/blog/frontier-code. Blog post.

Alexis Conneau, Min Ma, Simran Khanuja, Yu Zhang, Vera Axelrod, Siddharth Dalmia, Jason Riesa, Clara Rivera, and Ankur Bapna. FLEURS: Few-shot learning evaluation of universal representations of speech. In *2022 IEEE Spoken Language Technology Workshop (SLT)*, pp. 798–805, 2023.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, June 2022. URL http://arxiv.org/abs/2205.14135. arXiv:2205.14135 [cs].

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?, November 2025. URL http://arxiv.org/abs/2509.16941. arXiv:2509.16941 [cs.SE].

Peter K. Dunn and Gordon K. Smyth. Randomized quantile residuals. *Journal of Computational and Graphical Statistics*, 5(3):236–244, 1996.

Frank Garwood. Fiducial limits for the Poisson distribution. *Biometrika*, 28(3/4):437–442, 1936.

Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

GLM-5 Team. GLM-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. Evaluation of LLMs on syntax-aware code fill-in-the-middle tasks. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2403.04814.

Linyuan Gong, Alvin Cheung, Mostafa Elhoushi, and Sida Wang. Structure-aware fill-in-the-middle pretraining for code, 2025. URL https://arxiv.org/abs/2506.00204.

Yash Goyal, Tejas Khot, Douglas Summers-Stay, Dhruv Batra, and Devi Parikh. Making the V in VQA matter: Elevating the role of image understanding in visual question answering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6904–6913, 2017.

Sara Hooker. The Hardware Lottery, September 2020. URL http://arxiv.org/abs/2009.06489. arXiv:2009.06489 [cs.CY].

Hangrui Hu, Xinfa Zhu, Ting He, Dake Guo, Bin Zhang, Xiong Wang, Zhifang Guo, Ziyue Jiang, Hongkun Hao, Zishan Guo, Xinyu Zhang, Pei Zhang, Baosong Yang, Jin Xu, Jingren Zhou, and Junyang Lin. Qwen3-TTS technical report, 2026. URL https://arxiv.org/abs/2601.15621.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks, July 2026. URL https://arxiv.org/abs/2607.07946.

Atindra Jha, Naomi Sagan, Keisuke Kamahori, Irmak Sivgin, Rohan Sanda, Steven Gao, Mark Horowitz, Luke Zettlemoyer, Olivia Hsu, Jure Leskovec, Baris Kasikci, and Stephanie Wang. M*: A modular, extensible, serving system for multimodal models, 2026. URL https://arxiv.org/abs/2606.12688.

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?, November 2024. URL http://arxiv.org/abs/2310.06770. arXiv:2310.06770 [cs.CL].

Keisuke Kamahori, Shihang Li, Simon Peter, and Baris Kasikci. VibeServe: Can AI agents build bespoke LLM serving systems? *arXiv preprint arXiv:2605.06068*, 2026.

Marko Kojic, Ivan Bondyrev, Aral de Moor, Joseph Shtok, Petr Borovlev, Kseniia Lysaniuk, Madeeswaran Kannan, Ivan Dolgov, and Nikita Pavlichenko. Mellum2 technical report, 2026. URL https://arxiv.org/abs/2605.31268.

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*, pp. 611–626, 2023. URL http://arxiv.org/abs/2309.06180.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

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, 2025. URL https://arxiv.org/abs/2511.02230.

Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, Lichao Sun, Haotao Mi, and Leowei Liang. Long-Horizon-Terminal-Bench: Testing the limits of agents on long-horizon terminal tasks with dense reward-based grading. *arXiv preprint arXiv:2607.08964*, 2026.

Kung-Yee Liang and Scott L. Zeger. Longitudinal data analysis using generalized linear models. *Biometrika*, 73(1):13–22, 1986.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces, January 2026. URL http://arxiv.org/abs/2601.11868. arXiv:2601.11868 [cs.SE].

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-Lancer: Can Frontier LLMs Earn $1 Million from Real-World Freelance Software Engineering?, May 2025. URL http://arxiv.org/abs/2502.12115. arXiv:2502.12115 [cs.LG].

Moonshot AI. Kimi K3: Model card and evaluation results, 2026. URL https://huggingface.co/moonshotai/Kimi-K3/blob/main/README.md. Accessed September 20, 2026.

Ayush Nangia, Shikhar Mishra, Aman Gokrani, and Paras Chopra. ISO-Bench: Can coding agents optimize real-world inference workloads? *arXiv preprint arXiv:2602.19594*, 2026.

NVIDIA. GLM-5.3-Flash, 2026. URL https://docs.nvidia.com/nemo/automodel/model-coverage/vision-language-models/thudm/glm-5-3-flash. NeMo AutoModel documentation. Accessed September 23, 2026.

Gabriele Oliaro, Yichao Fu, May Jiang, Owen Lu, Junli Wang, Zhihao Jia, Hao Zhang, and Samyam Rajbhandari. FastKernels: Benchmarking GPU kernel generation in production. *arXiv preprint arXiv:2605.23215*, 2026.

OpenAI. OpenAI-MRCR: Multi-round co-reference resolution. Hugging Face dataset, 2025. URL https://huggingface.co/datasets/openai/mrcr.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, 2025.

Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. LibriSpeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5206–5210, 2015.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

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.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently Scaling Transformer Inference, November 2022. URL http://arxiv.org/abs/2211.05102. arXiv:2211.05102 [cs].

Proximal Labs. FrontierSWE: An ultra long-horizon coding agent benchmark, April 2026. URL https://www.proximal.ai/blog/frontierswe. Blog post.

Qwen Team. Qwen3-Omni technical report, 2025. URL https://arxiv.org/abs/2509.17765.

Qwen Team. Qwen3.6-27B. Hugging Face model card, 2026. URL https://huggingface.co/Qwen/Qwen3.6-27B. Accessed September 20, 2026.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 28492–28518, 2023.

Ben Rank, Hardik Bhatnagar, Ameya Prabhu, Shira Eisenberg, Karina Nguyen, Matthias Bethge, and Maksym Andriushchenko. PostTrainBench: Can LLM Agents Automate LLM Post-Training?, March 2026. URL http://arxiv.org/abs/2603.08640. arXiv:2603.08640 [cs.SE].

Vaibhav Srivastav, Steven Zheng, Eric Bezzam, Eustache Le Bihan, Nithin Rao Koluguri, Piotr Żelasko, Somshubra Majumdar, Adel Moumen, and Sanchit Gandhi. Open ASR leaderboard: Towards reproducible and transparent multilingual and long-form speech recognition evaluation, 2025. URL https://arxiv.org/abs/2510.06961.

Joshua Vendrow, Edward Vendrow, Sara Beery, and Aleksander Madry. Do large language model benchmarks test reliability?, 2025. URL https://arxiv.org/abs/2502.03461.

Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. KVCache cache in the wild: Characterizing and optimizing KV-Cache cache at a large cloud provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, 2025. URL https://arxiv.org/abs/2506.02634.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. FlashInfer-Bench: Building the virtuous cycle for AI-driven LLM systems. *arXiv preprint arXiv:2601.00227*, 2026.

Xinbo Xu, Ruihan Yang, Haiyang Shen, Wendong Xu, Bofei Gao, Ruoyu Wu, Kean Shi, Weichu Xie, Xuanzhong Chen, Ming Wu, Jason Zeng, Michael Heinrich, Elvis Zhang, Liang Chen, Kuan Li, and Baobao Chang. RoadmapBench: Evaluating long-horizon agentic software development across version upgrades. *arXiv preprint arXiv:2605.15846*, 2026.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-Bench: Can Language Models Rebuild Programs From Scratch?, May 2026. URL http://arxiv.org/abs/2605.03546. arXiv:2605.03546 [cs.SE].

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving, January 2025. URL http://arxiv.org/abs/2501.01005. arXiv:2501.01005 [cs].

Jehyeok Yeon, Ben Rank, and Maksym Andriushchenko. InferenceBench: A benchmark for open-ended LLM inference optimization by AI agents. In *Second Workshop on Agents in the Wild: Safety, Security, and Beyond*, 2026. URL https://openreview.net/forum?id=UNudgkp1Jp.

Peiqi Yin, Jiangyun Zhu, Han Gao, Chenguang Zheng, Yongxiang Huang, Taichang Zhou, Ruirui Yang, Weizhi Liu, Weiqing Chen, Canlin Guo, Didan Deng, Zifeng Mo, Cong Wang, James Cheng, Roger Wang, and Hongsheng Liu. vLLM-Omni: Fully disaggregated serving for any-to-any multimodal models, 2026. URL https://arxiv.org/abs/2602.02204.

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. USENIX Association, 2022. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Z.ai. GLM-5.3: Model card and evaluation results, 2026. URL https://huggingface.co/zai-org/GLM-5.3/blob/main/README.md. Accessed September 20, 2026.

Heiga Zen, Viet Dang, Rob Clark, Yu Zhang, Ron J. Weiss, Ye Jia, Zhifeng Chen, and Yonghui Wu. LibriTTS: A corpus derived from LibriSpeech for text-to-speech. In *Interspeech 2019*, pp. 1526–1530, 2019.

Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. WildChat: 1m ChatGPT interaction logs in the wild. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2405.01470.

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. *Advances in Neural Information Processing Systems*, 37, 2024. doi: 10.48550/arXiv.2312.07104. URL http://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*, 2024.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, Zixin Wen, Zhonglin Xie, Chenhui Gou, Linxuan Ren, Yueyang Wang, Junfeng Zhong, Rui Liu, Tian Gao, Yangguang Lin, Jingyuan Zhang, Maojia Song, Xuan Qi, Jinhong Wu, Chenyang Zhang, Yinzhu Piao, Ziru Niu, Hongbin Lin, Lingxiang Meng, Peng Tang, Chengyao Tang, Shanyu Wu, Huanyu Zheng, Yu Liu, Liya Zhu, He Wang, Ming Ding, Ziyu Wan, Hao Liu, Sibo Wang, Haotian Zhu, Xintian Zhang, Nan Chai, Yipeng Liu, Panhao Lai, Sihang Yuan, Zixin Su, Ge Zhang, Wangchunshu Zhou, Yantao Du, Wenhao Huang, and Guang Shi. EdgeBench: Unveiling Scaling Laws of Learning from Real-World Environments, July 2026. URL http://arxiv.org/abs/2607.05155. arXiv:2607.05155 [cs.CL].

Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, et al. NanoFlow: Towards optimal large language model serving throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 749–765, 2025a.

Xiaoyuan Zhu, Yaowen Ye, Tianyi Qiu, Hanlin Zhu, Sijun Tan, Ajraf Mannan, Jonathan Michala, Raluca Ada Popa, and Willie Neiswanger. Auditing black-box LLM APIs with a rank-based uniformity test, 2025b.

A TOKEN-LEVEL VERIFIER: FORMULATION AND CALIBRATION

This appendix specifies the calibration procedure and acceptance rules for the token-level verifier in Section 2.2.1. Throughout, τ, $K_l(N)$, κ, and ρ denote the thresholds used for acceptance, including the tolerance factor h. We set $h = 2$ throughout and use Φ for the standard normal CDF.

A.1 TRUSTED STACKS AND CALIBRATION RUNS

The trusted reference for token-level verification is Hugging Face Transformers, except for `agentic_coding`, which uses the vLLM build specified in Table 4. The reference supplies the comparison distributions; the production stacks below provide calibration traces that capture numerical variation during serving.

We use vLLM and SGLang as trusted implementations, selected before calibration. Each stack serves the task's traffic under the same arrival process and decoding contract as a graded submission. The emitted tokens are then teacher-forced through the reference. For greedy decoding, the reference runs deterministically at batch size one, and calibration records the flip rate and gap counts over repeated runs. Each stack must meet the required number of scored positions in Appendix A.3. For sampling, calibration records the reported and reference probabilities, with replies as the units for estimating standard errors.

A.2 CALIBRATION VARIABILITY AND PRECISION

For greedy decoding, calibration checks the reproducibility of the flip rate across runs. Let $\bar{r}$ and s be the mean and sample standard deviation of the flip rates from M independently seeded runs of one stack. We require

$$s\sqrt{(M-1)/\chi^2_{M-1,\,0.05}} \;\le\; 0.05\,\bar{r}, \tag{4}$$

where $\chi^2_{\nu,a}$ is the a-quantile of a chi-squared distribution with ν degrees of freedom. Under a normal approximation for the run-level rates, the left-hand side is a one-sided 95% upper confidence bound on their standard deviation. This criterion is evaluated once, after collecting the required number of scored positions over at least eight runs; calibration is accepted only if both stacks satisfy it.

For sampling, we use a clustered standard error of the mean divergence, with replies as clusters (Liang & Zeger, 1986). This allows dependence among tokens within each reply and assumes independence across replies. We require the standard error to be at most 5% of the mean divergence.

A.3 GREEDY CONTRACT

Flip-rate threshold. Let $\hat{r}_e$ be the mean flip rate of trusted stack e. The threshold is

$$\tau = h \max_e \hat{r}_e. \tag{5}$$

The tolerance factor specifies the allowed variation beyond the measured rates of the trusted implementations.

Gap-count thresholds. The cutoffs are selected from a half-nat grid starting at 0.5 nats. We retain the first cutoff, subsequent cutoffs at which the maximum per-stack pooled count decreases, and the first cutoff with no observed events. Let E_{el} be stack e's pooled number of gaps at least c_l over T_e

scored positions. Under a Poisson model for these counts, the one-sided 97.5% upper confidence bound on the event rate is (Garwood, 1936)

$$\lambda_{el}^{+} = \frac{\chi^2_{2(E_{el}+1),\,0.975}}{2T_e}, \qquad \lambda_l^{+} = \max_e \lambda_{el}^{+}. \tag{6}$$

For a graded trace of length N, define

$$\begin{aligned} b_l(N) &= \min\left\{k \in \mathbb{Z}_{\geq 0} : \Pr\big(\mathrm{Poisson}(\lambda_l^{+} N) > k\big) \leq \frac{\delta}{L+1}\right\}, \\ K_l(N) &= \lfloor h\, b_l(N) \rfloor, \qquad \delta = 10^{-3}. \end{aligned} \tag{7}$$

The nominal error budget is divided among the L gap-count conditions and the flip-rate condition. Calibration also sets a minimum trace length so that, under the same Poisson model, the flip-rate condition has rejection probability at most $\delta/(L+1)$ for all longer traces. These probabilities are model-based quantities used to construct the thresholds.

Calibration sample size. Write the confidence level of the upper bound above as $1-\beta$. At a cutoff with no observed events, this bound is $-\log(\beta)/T_e$. For a task-specific target rate μ per scored position, we require

$$T_e \geq \frac{-\log\beta}{\mu} \tag{8}$$

positions per stack. When the workload exceeds this target, calibration uses whole-session subsamples at the task's arrival rate.

Scored positions. The greedy counts include emitted tokens and stopping decisions. A reply that ends below its output cap without an explicit end-of-turn token is scored at one additional position using the reference's most probable end-of-turn token. Termination at the output cap adds no such position.

A.4 Sampling contract

Reported outcomes. At each position, let R_t contain the top-k tokens, except that tokens tied in reported probability with the $(k+1)$-st token are assigned to the *other* category. Thus $|R_t| \leq k$. The partition is determined from the report without inspecting the sampled token. Its outcomes are the singleton tokens in R_t and the complement $\mathcal{V} \setminus R_t$, where $\mathcal{V}$ is the vocabulary. The reference mass of each outcome is the sum of its constituent token probabilities. The reported probability of the *other* category is one minus the sum of the reported probabilities of tokens in R_t, after removing boundary ties. The emitted token is y_t; its grouped outcome is $x_t = y_t$ when $y_t \in R_t$ and $x_t = \mathrm{other}$ otherwise.

The KL divergence, PIT values, and log-probability residuals use this common partition. For full distributions $\tilde{p}_t, \tilde{q}_t$ and $C_t = \mathcal{V} \setminus R_t$, the chain rule gives

$$\mathrm{KL}(\tilde{q}_t \| \tilde{p}_t) = \mathrm{KL}(q_t \| p_t) + \tilde{q}_t(C_t)\,\mathrm{KL}(\tilde{q}_t(\cdot \mid C_t) \| \tilde{p}_t(\cdot \mid C_t))\,.$$

The omitted term measures disagreement within *other* and is nonnegative, so grouped KL is a lower bound on full KL. Calibration uses grouped KL.

KL-divergence threshold. Let $\hat{d}_e$ be trusted stack e's mean KL divergence and $\widehat{\mathrm{SE}}_e$ its standard error from Appendix A.2. We use an approximate one-sided 95% upper confidence bound and apply the tolerance factor:

$$\kappa = h \max_e \left(\hat{d}_e + \Phi^{-1}(0.95)\,\widehat{\mathrm{SE}}_e\right). \tag{9}$$

Fraction of tokens outside the top-k. An emitted token outside R_t is scored only as *other*, leaving its individual probability unresolved. We bound the fraction of these emissions. Let $\hat{\phi}_e$ and $\widehat{\mathrm{SE}}_{\phi,e}$ be this fraction and its standard error, clustered by reply, for trusted stack e. Define

$$\phi = \frac{1}{N}\sum_{t=1}^{N} \mathbf{1}[y_t \notin R_t], \qquad \rho = h \max_e \left(\hat{\phi}_e + \Phi^{-1}(0.95)\,\widehat{\mathrm{SE}}_{\phi,e}\right). \tag{10}$$

We require $\phi \leq \rho$ for acceptance.

End-of-turn counts. Deviations at a few positions can have little effect on the mean KL divergence. This can be problematic for low-probability end-of-turn tokens, since an early ending removes the rest of a reply. Let $\mathcal{E}$ be the set of end-of-turn tokens and π_t their total probability under the full reference distribution. For positions with $\pi_t \le \varepsilon$, define the observed ending count and the sum of reference probabilities as

$$O_\varepsilon = \sum_{t:\,\pi_t \le \varepsilon} \mathbf{1}[y_t \in \mathcal{E}], \qquad E_\varepsilon = \sum_{t:\,\pi_t \le \varepsilon} \pi_t. \tag{11}$$

We construct a Poisson-binomial distribution from the reference ending probabilities and reject when its probability of at least O_ε endings is at most α. This count uses the emitted tokens and the reference probabilities directly.

End-of-turn probability cutoff. The quantities O_ε and E_ε are defined in Eq. 11. We choose ε as the largest value on a logarithmic grid for which E_ε does not exceed the task's specified bound on the expected count in any calibration run. At grading time, define the comparison distribution and its upper-tail probability by

$$O'_\varepsilon = \sum_{t:\,\pi_t \le \varepsilon} B_t, \qquad B_t \overset{\text{ind}}{\sim} \text{Bernoulli}(\pi_t), \qquad p_{\text{end}} = \Pr(O'_\varepsilon \ge O_\varepsilon). \tag{12}$$

The B_t are independent Bernoulli variables with the reference probabilities observed along the trace. This check rejects unusually large ending counts and does not penalize small counts.

Randomized probability integral transform. For each emitted outcome x_t, we compute

$$u_t = F_{q_t}(x_t^-) + V_t\, q_t(x_t), \qquad V_t \sim \text{Unif}[0,1). \tag{13}$$

Here $F_{q_t}(x^-)$ is the reported mass preceding outcome x in the verifier's ordering, and the V_t are independent draws made by the verifier.

Sampling statistics and acceptance. For each emitted outcome, the log-probability residual is

$$s_t = \log q_t(x_t) + H(q_t), \qquad \mathbb{E}[s_t \mid q_t] = 0 \tag{14}$$

under faithful sampling, where H denotes entropy. Let $\mathcal{F}_{t-1}$ denote the history and the report available before the draw. When the conditional sampling distribution is q_t, the residual in Eq. 14 satisfies

$$\mathbb{E}[s_t \mid \mathcal{F}_{t-1}] = 0, \qquad \sigma_t^2 = \mathbb{E}[s_t^2 \mid \mathcal{F}_{t-1}] = \text{Var}_{X\sim q_t}[\log q_t(X)]. \tag{15}$$

We standardize the sum of residuals as

$$Z = \frac{\sum_{t=1}^N s_t}{\left(\sum_{t=1}^N \sigma_t^2\right)^{1/2}}. \tag{16}$$

When the total variance is zero, we set $Z = 0$ for acceptance; this check then provides no evidence for rejection. The log-probability test uses a normal approximation for Z, with two-sided critical value $\Phi^{-1}(1-\alpha/2)$. When tokens are sampled from their reported conditional distributions, the PIT values are independent and uniformly distributed on $[0,1)$ over a fixed number of positions. We assess their uniformity with the CvM test, using an approximate p-value. For a trace with fewer than two scored positions, we set $p_{\text{CvM}} = 1$ for acceptance. Empty traces are rejected. The same combination of randomized PIT values and a CvM test is used by Zhu et al. (2025b) to audit response scores against a distribution estimated from reference samples; here we apply it to individual tokens using the reported probabilities.

A nonempty trace with valid probability reports and ending decisions passes when

$$\begin{gathered} D_{\text{KL}} \le \kappa, \qquad \phi \le \rho, \\ p_{\text{CvM}} \ge \alpha, \qquad |Z| \le \Phi^{-1}(1-\alpha/2), \qquad p_{\text{end}} > \alpha, \end{gathered} \tag{17}$$

where $\alpha = 0.01$ is the nominal level of each statistical check. Sampling replies must end on an end-of-turn token or reach their output cap; ending below the cap without such a token fails the contract.

A.5 VERIFIER EVALUATION DETAILS

Controlled variants. Figure 5 evaluates Qwen3.6-27B servers with a configuration change or a wrapper that alters prompts or reply endings, alongside stock vLLM and SGLang. Both decoding contracts are evaluated on the `generic_serving` workload. Each uses thresholds calibrated from vLLM and SGLang under the corresponding contract, with greedy flip-severity ceilings computed for each trace's length. There is one run per variant, except that sampling results pool three runs each for stock vLLM and SGLang and eight each for forced endings at 4096 and 8192 tokens. The ending-count check uses one run each of vLLM, SGLang, temperature 0.2, and the forced endings at 4096 and 8192 tokens.

Forced endings replace the token at 1024, 2048, 4096, or 8192 tokens with an end-of-turn token and, under sampling, report the model's own probability of that token. At 8192 tokens, this cuts 0.76% of greedy replies and 1.6% of sampling replies (across eight runs). Five additional sampling variants end replies below their output cap without an end-of-turn token; all fail the termination rule and are omitted from the plot.

Plot coordinates. Each axis takes the largest normalized value among its checks. Counts, rates, and divergences are divided by their thresholds. For the uniformity (CvM) and ending-count checks, we plot $\log(1/p)/\log(1/\alpha)$, which maps the p-value threshold α to one. The log-probability residual uses $|Z|/\Phi^{-1}(1-\alpha/2)$. Under sampling, *reference agreement* groups mean KL divergence, the ending-count test, and the fraction of tokens outside the top-k; *sampling consistency* groups the uniformity and log-probability tests.

InferenceBench comparison. For Table 3, InferenceBench's verifier evaluates 500 MMLU-Pro questions with four requests in flight and accepts at least 95% of reference accuracy. We run it twice per defective server and report the lower relative accuracy. Our verifier results come from each server's original task evaluation; Appendix C documents the six defects.

A.6 LIMITATIONS

The verifier combines statistically motivated checks with empirical calibration against trusted serving stacks. We evaluate its behavior on benign implementation variants, controlled deviations, and concrete defects in agent-generated servers (Section 4.3). These experiments support its ability to distinguish ordinary numerical variation from implementation errors in the evaluated settings.

The greedy contract bounds the frequency and severity of flips across a trace. It does not constrain which positions contain those flips, and an allowed flip may end a reply early. The sampling KL threshold applies to the reported distribution on the top-k tokens and the *other* category. Differences within that category are not resolved. The emission tests assess aggregate consistency with these reports; passing them does not establish the conditional sampling distribution at every position. The nominal test levels do not specify an overall false-rejection probability for the combined verifier with output-dependent trace lengths.

B EVALUATION ENVIRONMENT

Agents receive a live GPU environment, a task specification, and development resources for building and testing their servers. For a given task, all agents receive the same hardware allocation, software environment, and access to reference implementations. Scored evaluation uses a fresh container and a withheld workload.

B.1 HARDWARE

Experiments run on NVIDIA HGX H200 systems with 141 GB of device memory per GPU and NVLink connectivity between GPUs. As listed in Table 1, `agentic_coding` uses eight GPUs, `omni_chat` uses two, and the remaining tasks use one each. Each run receives 180 GB of host memory. Agents can execute and profile their implementations on the allocated GPUs throughout the build.

B.2 Task Specification

The task specification identifies the model to serve, hardware allocation, workload, correctness criteria, and performance objective. It describes request arrival patterns, session structure, decoding rules, output limits, and latency targets. It also defines the serving interface, including streaming responses and the token IDs or log-probability reports required by the verifier. Agents choose how to implement model execution, memory management, batching, scheduling, and parallelism within this contract.

B.3 Development Data and Reference Access

Model weights, configurations, and tokenizers or processors are available locally and mounted read-only. Agents receive development data and the prompt templates needed to exercise each workload; the scored test split is withheld from the build environment. Correctness criteria are disclosed, and every task provides access to a reference implementation for self-checks. For `agentic_coding`, reference checks use a separate vLLM service.

Speech and omni tasks additionally provide reference outputs and correctness scores for development examples. The deep-research task provides a research loop, retrieval tools, and a development corpus so agents can test complete trajectories. During scored evaluation, retrieval runs separately under evaluator control. These resources support local iteration without exposing the scored workload.

B.4 Implementation and Profiling Tools

The environment supports Python, C++, and CUDA, including PyTorch compilation and native extensions. Agents can write GPU kernels with Triton, CUTLASS/CuTe, or cuTile, and use numerical and communication libraries such as cuBLAS, cuDNN, and NCCL. Task environments include the model-loading and modality-specific dependencies needed for text, image, or audio processing. Nsight Systems, Nsight Compute, Compute Sanitizer, and CUDA-GDB are available for performance analysis and debugging.

B.5 Network and Dependency Restrictions

In the from-scratch setting, agents cannot browse the Internet or retrieve external repositories. Package installation is allowed, but high-level inference engines and optimized inference-kernel packages are excluded, including vLLM, SGLang, TensorRT-LLM, FlashAttention, FlashInfer, and xFormers. The numerical libraries and kernel-development tools above remain available. These restrictions apply during both development and installation of the final submission.

B.6 Build Budget and Final Evaluation

Main evaluations allow 12 hours per build; the time-budget ablation varies this limit. Agents receive remaining-time feedback and are instructed to continue improving their implementations while time remains. Native goal mode in Codex and Claude Code supports this continued optimization.

The final submission is the latest committed implementation, including its declared dependencies and launch entry point. It is installed and launched in a fresh copy of the task environment, so evaluation does not rely on uncommitted files or undeclared dependencies from the build session. A separate workload generator sends requests to the server and records its outputs and timings. Correctness and performance are assessed from this same run; submissions that fail correctness receive a score of zero.

C LOGIC ERRORS IN DEFECTIVE SERVERS

This appendix explains the logic errors in the six servers in Table 3. A key–value (KV) cache stores past attention keys and values; recurrent layers instead carry a state updated as tokens are processed. A hybrid model uses both, so cached attention and recurrent states must represent the same token history. Prefix caching reuses these states when requests share an initial token sequence. The excerpts below retain the operations relevant to each error and omit unrelated code.

C.1 UNTRACKED ATTENTION TENSOR LIFETIMES ACROSS CUDA STREAMS

CUDA streams execute queued GPU operations asynchronously. Here, `attend` allocates each request's attention output on a separate stream, and `torch.cat` reads those outputs on the main stream. The `wait_stream` calls ensure that attention finishes before concatenation starts. They do not protect the tensors after concatenation is queued: the code releases them without recording the main stream's use with `record_stream`, or making their allocation streams wait for that use to finish. The lifetime requirement is that the output storage remain valid until the asynchronous concatenation has finished reading it.

Python

```python
producer_stream = torch.cuda.current_stream()
for batch_index, cache in enumerate(caches):
    stream = attention_streams[batch_index]
    stream.wait_stream(producer_stream)
    with torch.cuda.stream(stream):
        outputs.append(attend(batch_index, cache))
for stream in attention_streams[:len(caches)]:
    producer_stream.wait_stream(stream)
attention_output = torch.cat(outputs, dim=0)
```

C.2 SLOT COMPACTION ENDS CO-RUNNING REPLIES

Active requests occupy consecutive slots. A decode step produces one token per request in `toks`, indexed by that slot order. When a request finishes, `_finish(s, "stop")` moves the last active request into slot `s`, but does not move its token in `toks`. The `continue` revisits the same slot and reads `toks[s]` again, assigning the finished request's end-of-sequence token to the moved request. This mismatch between request slots and token indices causes premature termination and can cascade through the remaining requests.

Python

```python
toks = self.decode_step()
s = 0
while s < self.n_active:
    req = self.slots[s]
    tok = toks[s]
    # ...
    stop = tok in req.stop_ids
    # ...
    if stop:
        self._finish(s, "stop")
        continue  # slot s now holds a moved req; re-check same index
```

C.3 RACE IN THE RECURRENT KERNEL'S REPLAY

Speculative decoding proposes tokens before verifying them; this kernel replays the accepted tokens to update the recurrent state. It saves the gate inputs that control each token's state update in `ASAVE` and `BSAVE`. Each head's value columns are split between two CUDA thread blocks (Triton kernel instances), selected by `tl.program_id(2)`. Both blocks read the same `ASAVE` and `BSAVE` entries, then overwrite them with the current step's inputs. Those addresses do not depend on the value-column tile, and there is no synchronization between the blocks. One block can therefore

overwrite the gates before the other loads them, applying the wrong token's gates to part of the recurrent state.

Python / Triton

```
vb = tl.program_id(2)  # Selects the value-column tile, not the gate entries.
for t in range(RL):  # Replay previously accepted tokens.
    # ...
    ar = tl.load(ASAVE + slot * sqb + t * sqt + h).to(tl.float32) + dtb
    br = tl.load(BSAVE + slot * sqb + t * sqt + h).to(tl.float32)
# ...
for t in range(QL):  # Process the current step and save its gate inputs.
    # ...
    if RL > 0:
        tl.store(ASAVE + slot * sqb + t * sqt + h, arr)
        tl.store(BSAVE + slot * sqb + t * sqt + h, brr)
```

C.4 CUDA-GRAPH PADDING ROWS OVERWRITE LIVE KV

A captured CUDA graph executes a fixed batch shape. When fewer requests are active, the server fills unused rows with dummy inputs, but still executes their KV writes. Each row's block table maps token positions to physical cache pages. Initialization clears only its first two entries (`btab_cpu[rr, :2]`), leaving later mappings from previous requests. Decode steps advance dummy positions as well as live positions. Once a dummy row reaches an uncleared entry, its writes can overwrite a live request's cached keys and values.

Python

```
# B: active requests; gb: captured batch size; nt: rows per request.
for i in range(B, gb):
    for j in range(nt):
        rr = nt * i + j
        self.ids_r[rr] = 1
        self.pos_r[rr] = j
        self.ctx_r[rr] = j + 1
        self.btab_cpu[rr, :2] = 0
```

C.5 PREFIX-CACHE HIT RESUMES A MISALIGNED STATE

This server processes prompt tokens in tiles of `BT` tokens and caches state for reuse by requests with a shared prefix. The intended snapshot offset within the current prefill chunk is `off`. Attention uses that exact offset, but the recurrent snapshot uses `off // BT`, rounding down to a tile boundary. When the offset is not aligned, the recurrent state omits the remaining tokens before the intended snapshot. A cache hit nevertheless resumes after the full prefix, so those recurrent updates are skipped rather than recomputed.

Python

```
snap_pos[nd + pi] = off
snap_cslot[nd + pi] = sidx
snap_chunk[pi] = off // BT
snap_slot[pi] = sidx
```

In the kernel, `sc` is the stored tile count and `i_c` is the current tile index. The state is saved after processing tile `sc - 1`, covering only `sc * BT` tokens:

Python / Triton

```
if DO_SNAP:
    if i_c == sc - 1:
        tl.store(SNAP + ss * stride_sb + i_h * stride_sh
                 + rk[:, None] * stride_sk + rv[None, :], state)
```

C.6 PREFIX CACHE WITH SPECULATION SKIPS A KV BLOCK

The vLLM-based server must choose a reusable prefix length supported by both its attention and recurrent caches. With speculation enabled (`use_eagle`), attention lookup removes its last cached block with `computed.pop()` so that block will be recomputed. Recurrent lookup ignores this adjustment: its search bound still uses the original `max_length` and can return a hit one block longer. The coordinator accepts that longer prefix and skips the recomputation needed by attention. Attention then reads a zero-filled KV block in place of the missing cached state. The two cache lookups disagree about how much of the prefix is ready to reuse.[1]

Python

```python
# Full-attention lookup
if use_eagle and computed_blocks[0]:
    for computed in computed_blocks:
        computed.pop()
# Recurrent lookup
max_num_blocks = max_length // block_size
for i in range(max_num_blocks - 1, -1, -1):
    if cached_block := block_pool.get_cached_block(
        block_hashes[i], kv_cache_group_ids
    ):
        # ...
        break
```

[1] Proposed upstream fix: `https://github.com/vllm-project/vllm/pull/48375`.

D Full Task Specifications and Results

D.1 Task Specifications

Table 4 lists each task's served model, hardware, baseline stack, SLO, correctness gate and decoding rule, and Table 5 characterizes its traffic. The baselines are vLLM (Kwon et al., 2023) and vLLM-Omni (Yin et al., 2026), at the versions that measured the baseline credit B (Section 2.3). The workload statistics come from one baseline drive per task: one pass of the complete test workload against the baseline stack, whose own replies determine the output lengths. Tokens are counted with each model's tokenizer or processor. A request's input tokens are the full sequence the server prefills, including the chat template and the placeholder tokens that represent images and audio. Its output tokens are the tokens the baseline generated, including the end-of-turn token. The prefix share is the fraction of all input tokens in the drive that match a prefix of a sequence seen earlier in the same drive: an earlier request's input, or that input followed by its reply once the reply has completed. This is the fraction an unbounded prefix cache could reuse; an image or audio clip matches only itself. Turns are counted per session, and the traffic span runs from the first request sent to the last reply received, which excludes server start-up.

Table 4: **Task serving specifications.** Precision is that of the served checkpoint. The baseline is the production stack that measured the baseline credit B. The SLO is per request, except for `deep_research`, where it applies to each trajectory.

Task	Model	GPUs	Baseline	SLO	Correctness gate	Decoding
`generic_serving`	Qwen3.6-27B (Qwen Team, 2026) (bf16)	1×H200	vLLM 0.22.1	TTFT 300–1,000 ms, TPOT 50 ms (per tenant)	Greedy contract (§2.2.1)	greedy
`agentic_coding`	GLM-5.3-Flash (GLM-5 Team, 2026) (fp8)	8×H200	vLLM 0.29.1rc1 dev[a]	TTFT $1.4\,\mathrm{s} + 8\,\mu\mathrm{s}$ per prompt token, TPOT 20 ms	Sampling contract (§2.2.1)	sampled
`deep_research`	Gemma-4-31B-it (Gemma Team, 2026) (bf16)	1×H200	vLLM 0.28.0	Trajectory makespan 1,800 s	Sampling contract (§2.2.1)	sampled
`code_completion`	Mellum2-12B-A2.5B-Base (Kojic et al., 2026) (bf16)	1×H200	vLLM 0.22.1	TTFT 200–600 ms, TPOT 10 ms (per tenant)	Greedy contract (§2.2.1)	greedy
`speech_recognition`	Whisper-large-v3 (Radford et al., 2023) (fp16)	1×H200	vLLM 0.28.0	Request latency 6 s	Mean $1 - \mathrm{WER}$ (Srivastav et al., 2025) against the reference (§2.2)	greedy
`speech_synthesis`	Qwen3-TTS-12Hz-1.7B-CustomVoice (Hu et al., 2026) (bf16)	1×H200	vLLM-Omni 0.28.0	Time to last audio frame 10 s	Sampling contract (§2.2.1); log-mel PSNR at least 29 dB	sampled
`omni_chat`	Qwen3-Omni-30B-A3B-Instruct (Qwen Team, 2025) (bf16)	2×H200	vLLM-Omni 0.28.0	Request latency 15 s	Word F1 (image) and BLEU (speech) (Qwen Team, 2025) against the reference (§2.2)	greedy

[a] The vLLM nightly build of 2026-09-19 (0.29.1rc1.dev397, commit `a8d1aa9c`); the same build is the correctness reference for this task.

Table 5: **Task workloads.** Statistics are computed from one baseline drive per task. Token pairs give the median and 99th percentile per request (p50 / p99). Input tokens are the full prompt the model prefills, media placeholders included, and output tokens are those the baseline generated. Turns give the median and maximum per session. For the two multi-tenant tasks, the task row pools all requests and an indented row follows for each tenant; a tenant's prefix share counts matches against the whole drive.

Task	Source	Load	Tenants	Input tokens (p50 / p99)	Output tokens (p50 / p99)	Prefix share	Turns
`generic_serving`	One public dataset per tenant; tenant mix fitted to Bailian trace statistics (Wang et al., 2025)	Gamma, 1.11 req/s 1,317 req., 1,710 s	4	783 / 4,801	440 / 7,110	63%	1 / 8
`chat`	WildChat (Zhao et al., 2024)	Gamma, 0.084 sess./s 201 req.		170 / 6,887	2,530 / 8,901	34%	2 / 6
`api_short`	BFCL (Patil et al., 2025)	Gamma, 0.666 sess./s 839 req.		737 / 3,044	268 / 1,832	57%	1 / 1
`reasoning`	GSM8K-Platinum (Vendrow et al., 2025)	Gamma, 0.133 sess./s 199 req.		787 / 855	1,795 / 3,548	91%	1 / 1
`long_session`	MRCR (OpenAI, 2025)	Gamma, 0.015 sess./s 78 req.		3,040 / 5,059	2,057 / 7,952	90%	3.5 / 8
`agentic_coding`	Terminal-Bench 4.0 tasks (Merrill et al., 2026), run live	Poisson, 0.050 sess./s 1,577 req., 2,787 s	1	80.8k / 601k[b]	534 / 27.9k	98%	46.5 / 165
`deep_research`	BrowseComp-Plus (Chen et al., 2025), live retrieval	Poisson, 0.015 sess./s 683 req., 3,692 s	1	13.2k / 73.0k	745 / 8,192	82%	13 / 41
`code_completion`	Real-FIM-Eval typing sessions and SAFIM infills	Poisson, 24.6 req/s 26,317 req., 1,105 s	4	695 / 7,061	23 / 256	45%	1 / 8
`commit_add`	Real-FIM-Eval, Add (Gong et al., 2025)	Poisson, 1.14 sess./s 6,440 req.		2,337 / 7,380	70 / 256	51%	6 / 8
`commit_edit`	Real-FIM-Eval, Edit (Gong et al., 2025)	Poisson, 1.16 sess./s 6,477 req.		2,160 / 7,816	55 / 256	45%	5 / 8
`algo_block`	SAFIM, block (Gong et al., 2024)	Poisson, 6.15 req/s 6,700 req.		299 / 2,723	20 / 256	27%	1 / 1
`algo_control`	SAFIM, control flow (Gong et al., 2024)	Poisson, 6.15 req/s 6,700 req.		312 / 2,710	8 / 256	25%	1 / 1
`speech_recognition`	LibriSpeech test-clean (Panayotov et al., 2015)	Poisson, 52 req/s 2,400 req., 47 s	1	4 / 4[c]	21 / 77	0%	1 / 1
`speech_synthesis`	LibriTTS dev-clean (Zen et al., 2019)	Poisson, 2 req/s 2,400 req., 1,221 s	1	29 / 80[d]	69 / 303	34%	1 / 1
`omni_chat`	VQAv2 (Goyal et al., 2017); FLEURS fr→en (Conneau et al., 2023)	Poisson, 30 req/s 4,600 req., 154 s	1	224 / 427[e]	7 / 48	50%	1 / 1

[b] Six of the 30 sessions resume recorded trajectories at 200k, 400k or 600k prompt tokens. [c] Input tokens are the decoder's four task tokens. The encoder input is fixed at 1,500 frames covering a 30 s window, which holds each request's clip of 5.7 / 25.9 s (p50 / p99). Output tokens are the transcript. [d] Input tokens are the talker's prefix: the sentence's 18 / 69 text tokens and 11 fixed positions, whose first nine, shared by every request, make up the prefix share. Output tokens are codec frames of 16 codes and 80 ms each, 5.5 / 24.2 s of audio. [e] Half the requests carry an image and half a clip of French speech. Input tokens are 26 / 35 text tokens plus the image's 260 / 400 placeholder tokens or the clip's 145 / 281, for clips of 11.2 / 21.6 s. The 2,306 image requests use 452 distinct images, so the prefix share is 78% on the image requests and 2% on the speech requests.

D.2 SAMPLE BUILD BRIEF

The following abridged template illustrates the build brief for `generic_serving`. Before a run, the harness renders the Jinja placeholders using the task specification and calibrated verifier thresholds. The tenant loop expands into a workload description for each tenant. The toolchain inventory is abbreviated as a section placeholder.

Markdown / Jinja2

```
# Task Description

## Goal
Build a hyperoptimized serving stack for `{{ model.base_checkpoint }}`
under the workload below. Optimize until you reach the physical limits
of the hardware or exhaust your budget. The grader drives all tenants
concurrently against one instance of your server.

Your score is your speedup over a production serving framework in
SLO-normalized latency. For every request:

    r = max(TTFT / that tenant's TTFT budget,
            mean TPOT / that tenant's TPOT budget)
    score = exp(mean over requests of log(1/r))
            / the same quantity for the production framework
              on this workload

Each turn of a multi-turn session is a separate request, sent and
scored on its own. Each such request carries the conversation so far.
Your implementation must not fail any request. Every request must
receive a complete streaming response.

Correctness is checked at the token level: the exact token IDs you
serve are teacher-forced through a bf16 reference implementation of
this model.
{% if verification %}
The run fails if more than {{ verification.flip_rate_pct }}% of your
served tokens have a lower reference log-probability than the
reference's most likely token.
{% if verification.flip_curve_levels %}
The run also fails if the number of such tokens exceeds a calibrated
ceiling at any of several log-probability gap levels. The ceilings
scale with the run's total token count.
{% endif %}
Exact ties do not count as deviations.
{% endif %}
The check is pooled over the whole run, not performed per tenant.
Serve greedy: the argmax of the raw logits, with none of the
checkpoint's generation_config sampling defaults applied.

## Model
- Checkpoint: {{ model.base_checkpoint }}
- Chat format: {{ model.chat_format }}

## Hardware Envelope
{{ hardware.accelerator_count }}x{{ hardware.accelerator_kind }}
{{ hardware.memory_gb_per_device }} GB/device

## Workload
Offered load: {{ offered_load }} requests/s, an upper bound since
a session can end before its last turn.
{% for tid, wf, ds in tenants_data %}
### Tenant {{ tid }}
- Dataset: {{ (ds.dataset.path or ds.dataset.repo if ds else None) or "(none)" }}
- Sessions: {{ wf.sessions }}
- Requests: {{ wf.requests }}
{% if wf.think_time %}
- Between turns: {{ wf.think_time }}
{% endif %}
- Per-request output cap: {{ wf.output_cap }} tokens
- SLO: {{ wf.slo }}
{% endfor %}

## Network Contract
GET /health: return 200 only when the model is fully loaded and
the inference endpoint can serve a request immediately.
### POST /v1/chat/completions
Accept OpenAI chat/completions requests and return chat completion responses.
The harness sends `max_tokens` on every request body; treat it as a binding
per-request output cap. Do not substitute a server-side default when the
```

````
request supplies `max_tokens`.

Return a `token_ids` array on each choice (`choices[].token_ids`): the integer
vocabulary IDs of the exact tokens you emitted (NOT a re-tokenization of the
text). This is mandatory on every response.
Support streaming: when the request sets `"stream": true`, respond with an OpenAI
SSE stream (`data: {chunk}` lines, `chat.completion.chunk` objects with `delta`,
terminated by `data: [DONE]`). Include the per-chunk `token_ids` array for that
chunk's tokens.
Request:
```json
{
  "model": "served-model",
  "messages": [
    {
      "role": "user",
      "content": "<prompt>"
    }
  ],
  "return_token_ids": true
}
```
Response:
```json
{
  "choices": [
    {
      "message": {
        "role": "assistant",
        "content": "<answer>"
      },
      "finish_reason": "stop",
      "token_ids": [
        1234,
        5678
      ]
    }
  ],
  "usage": {
    "completion_tokens": 2
  }
}
```

## Build Instructions
Write your server as `{{ server_path }}`, run as
`python {{ server_path }} --host <H> --port <P>`.
The entry point may execute or bind compiled implementations.
A development split is at `{{ dev_rows_path }}`, one per tenant;
the scored test split is withheld.

## Toolchains and Libraries
{{ toolchain_section }}

## Submission
You MUST work in a git repository at the repo root. Only the latest commit
on `main` is evaluated; uncommitted or unmerged work is discarded. Merge to
`main` the moment you have a correct server that performs better than your
last commit, and again after every such improvement -- not only at the end.
The repo must contain:
- `pyproject.toml` declaring every dependency (the harness installs with
  `pip install .`; undeclared packages are absent at eval time).
- `server.py`, launched as `python server.py`.
The harness builds your submission with `pip install .` then `python
server.py` in a fresh copy of this environment.

## Your budget
You have 12 hours of wall-clock time for this build. Call `get_budget` (the
`budget` MCP server) periodically to see how much remains, and plan your
remaining work to stay within it. Leave headroom to commit working code
before you run out.
````

D.3 Cost Analysis

Table 6 reports estimated build-phase costs for the 126 main runs and 24 additional ablation runs. Since we utilized in-house clusters for the GPU sandboxes, we estimate GPU costs using Amazon EC2 `p5e.48xlarge` Capacity Blocks pricing in US West (Oregon): $47.76 per instance-hour for eight H200 GPUs, equivalent to $5.97 per GPU-hour.[2] For each run, we multiply its fixed build budget by its allocated GPU count and this per-GPU rate. The allocation is eight GPUs for `agentic_coding`, two for `omni_chat`, and one for every other task. This prorates the instance price across its GPUs. Every main run is charged for the full 12-hour budget, regardless of its actual duration, giving 540 GPU-hours and $3,223.80 per model across 21 runs. Ablations use their assigned build budgets: 12 hours for the reasoning-effort and starting-point experiments, and 3, 6, 24, or 36 hours for the additional time-budget runs.

Table 6: **Build-phase costs (USD).** Each model row covers seven tasks with three repeats. Ablation rows contain only additional Opus 5 builds; shared main runs are counted once. GPU costs charge the full allocated build budget. Row totals are computed before rounding.

Model / experiment	API cost	GPU cost	Total
Main rollouts (21 builds per model at 12 hours)			
Opus 5	$5,199.71	$3,223.80	$8,423.51
Opus 4.8	$5,563.12	$3,223.80	$8,786.92
GPT 5.6 Sol	$3,896.72	$3,223.80	$7,120.52
GPT 5.5	$2,448.79	$3,223.80	$5,672.59
Kimi K3	$1,271.76	$3,223.80	$4,495.56
GLM 5.3	$2,324.92	$3,223.80	$5,548.72
Ablations (Opus 5)			
Reasoning effort (9 builds)	$1,536.08	$644.76	$2,180.84
Time budget (12 builds)	$4,075.48	$1,235.79	$5,311.27
Starting point (3 builds)	$550.75	$214.92	$765.67

D.4 Validation During Incremental Improvement

All three vLLM-based builds in Section 4.1.3 reuse vLLM's multi-token prediction (MTP) implementation. One failed run retains a scheduler change that corrupts recurrent state when requests are waiting for admission; its four-prompt sequential comparison does not exercise this condition. The other checks over 330,000 tokens under concurrency, but samples isolated turns rather than growing conversations, missing a prefix-cache defect already present in vLLM (Appendix C.6). In the passing run, token checks on concurrent, multi-turn sessions lead the agent to disable prefix caching and revert a change that skips MTP drafting during prefill.

[2] Amazon EC2 Capacity Blocks for ML pricing, accessed September 20, 2026.