SERVINGBENCH: CAN AGENTS DELIVER DAY-0 MODEL SUPPORT IN SERVING ENGINES?

Anonymous authors
Paper under double-blind review

ABSTRACT

Serving engines such as vLLM now provide day-0 model support, making major open-weight models available on the day they are released. However, supporting a new model correctly and efficiently on day 0 can be non-trivial. Engineers must implement the new model architecture, integrate it with the serving runtime, and, for some models, optimize kernels and backend code for the target accelerator. We therefore ask: can coding agents deliver correct and efficient day-0 model support? To answer this question, we introduce SERVINGBENCH, a benchmark of 148 day-0 model support tasks across 10 accelerator tiers from NVIDIA, AMD, and Google. An automated pipeline constructs executable environments from selected model-integration pull requests in vLLM, SGLang, and TensorRT-LLM. Each environment includes correctness checks and an expert performance baseline validated on the target accelerator. Evaluating seven state-of-the-art models, we find that agents resolve 56.7% of tasks on average, but 33% of resolved patches serve more slowly than the engine maintainers' implementation, by a median of 18%. Even the strongest model in our evaluation (i.e., Claude Opus 5) delivers a correct patch at the expert's serving speed on only 52.7% of tasks, and 32% of its resolved patches are slower, some at about half the expert's speed. The pipeline can incorporate new model releases, allowing SERVINGBENCH to track progress on a continually evolving systems engineering task.

1 INTRODUCTION

Open-weight models are deployed through serving engines such as vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024), and TensorRT-LLM (Liebenwein et al., 2026). To let users run new models right away, the major serving engines now aim to support popular models on their release day, which we call *day-0 model support*. From the end of 2024 to August 2026, the number of model architectures supported by vLLM and SGLang grew 3.3×, and the number of accelerator platforms they support grew 2.2× (vLLM Team, 2026e; SGLang Team, 2026).

However, adding day-0 model support to a serving engine is non-trivial. A model release provides weights and a reference implementation, such as code for Hugging Face Transformers (Wolf et al., 2020). The reference defines what the model computes, but efficient serving requires integration with the engine. Consider the vLLM pull request (PR) that added DeepSeek-V4, which involved 15 contributors and changed roughly 17,000 lines across 150 files (vLLM Contributors, 2026). Engineers first reimplement the reference model using the engine's layers and load the released weights. They then integrate the model with tensor parallelism, the KV cache, and request handling for prefill and decode. Common operations reuse existing kernels, while DeepSeek-V4's new operators and FP8 and FP4 formats require specialized CUDA and Triton kernels (vLLM Team, 2026b). These kernels must match the engine's data layouts and run efficiently within its serving runtime. Finally, engineers verify the outputs against the reference and measure end-to-end serving performance.

Automating this work could help serving engines keep pace with new model releases. Coding agents built on large language models (LLMs) can already resolve GitHub issues and implement repository-level features (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025; Zhang et al., 2024; Xia et al., 2025). We therefore ask: *can coding agents deliver correct and efficient day-0 model support?*

Existing benchmarks answer this question only in part (Table 1). FEA-Bench (Li et al., 2025) and FeatureBench (Zhou et al., 2026) focus on feature development in general software repositories

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.

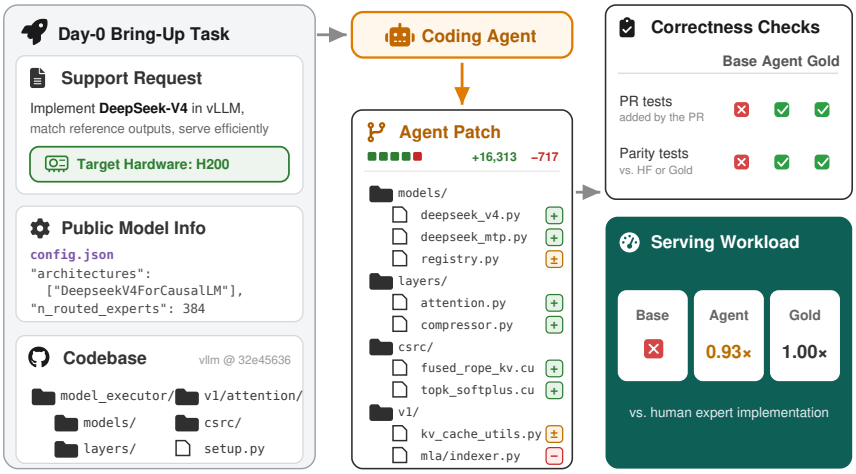


Figure 1: **A SERVINGBENCH task.** Given the engine before integration, the released checkpoint, and reference code, an agent adds support for the new model. Hidden checks verify correctness on the target accelerator. Serving speed is then compared with Gold, the maintainers' implementation.

and grade it with unit tests. SWE-fficiency (Ma et al., 2026), KernelBench (Ouyang et al., 2025), and InferenceBench (Yeon et al., 2026) focus on performance, but they optimize code that already works, such as Python libraries, PyTorch programs, and inference servers for supported models. SWE-Serve (Williams et al., 2026) is the closest to our setting, but most of its tasks are bug fixes or other inference features in SGLang rather than model support. Its GPU tasks use a single accelerator type (NVIDIA H100), and only three tasks include performance checks.

To address this gap, we develop a four-stage pipeline that automatically constructs executable model-integration environments across diverse accelerator types (Figure 2). For each selected PR, the pipeline reconstructs the pre-integration engine, validates correctness with reusable behavioral checks, and measures serving speed against the maintainers' implementation. This allows the pipeline to incorporate new model releases without designing a new evaluation for each task.

Using this pipeline, we construct SERVINGBENCH, a benchmark of 148 model bring-up tasks from vLLM, SGLang, and TensorRT-LLM across 10 accelerator tiers from NVIDIA, AMD, and Google. Each task rewinds the engine to just before its first model-support PR and asks an agent to add the missing support from the released checkpoint and reference implementation (Figure 1), recreating the original model bring-up problem before an upstream implementation existed.

We evaluate seven state-of-the-art models using mini-SWE-agent (Lieret & Jimenez, 2025) and find a substantial gap between generating a correct patch and delivering efficient model support. Agents resolve only 56.7% of tasks on average, and 33% of resolved patches serve more slowly than the maintainers' implementation, by a median of 18%. Even Claude Opus 5, the strongest model, delivers correct support at the expert's serving speed on only 52.7% of tasks, with some correct patches serving at roughly half the expert's speed. Kernel work and larger bring-ups remain particularly challenging. Overall, our contributions are:

- An automated pipeline for constructing model-integration environments across diverse accelerators, enabling evaluation of future model releases and providing a mechanism that could be used to create new agent training environments.
- SERVINGBENCH, with 148 tasks across three serving engines and 10 accelerator tiers, evaluating checkpoint correctness and serving performance.
- An evaluation of seven coding agents that quantifies the gap to expert serving performance and identifies failures in model implementation, runtime integration, and kernel work.

	Task			Evaluation			Reference patch (mean)			
Benchmark	Repo-level	New capability	Model support	Performance	Hardware	Tasks	Files edited	Lines added	New functions	New classes
SWE-bench (Jimenez et al., 2024)	✓	✗	✗	✗	CPU	2,294	1.7	27	0.7	0.1
FEA-Bench (Li et al., 2025)	✓	✓	✗	✗	CPU	1,401	3.4	123	4.5	0.8
FeatureBench (Zhou et al., 2026)	✓	✓	✗	✗	CPU, NVIDIA GPU	200	14.4	759	29.9	0.1
SWE-fficiency (Ma et al., 2026)	✓	✗	✗	✓	CPU	498	2.2	32	0.9	0.2
KernelBench (Ouyang et al., 2025)	✗	✗	✗	✓	NVIDIA/AMD GPU	250	–	–	–	–
InferenceBench (Yeon et al., 2026)	✗	✗	✗	✓	NVIDIA GPU	4	–	–	–	–
SWE-Serve (Williams et al., 2026)	✓	✓	(✓)	(✓)	CPU, NVIDIA GPU	53	10.1	828	25.4	4.4
SERVINGBENCH	✓	✓	✓	✓	NVIDIA/AMD GPU, TPU	148	10.7	1,485	45.4	11.9

Table 1: **Comparison with related benchmarks.** Marks follow each benchmark's paper or release; (✓) indicates partial coverage. *New capability* denotes tasks introducing new functionality rather than fixing or optimizing existing code; *Model support* denotes tasks adding a new model architecture to a serving engine; and *Hardware* lists target processors. The last four columns give the mean files edited, lines added, and new Python functions and classes per released reference patch.[2]

2 RELATED WORK

Table 1 compares SERVINGBENCH with the closest benchmarks by task, evaluation, and reference patch size.

Repository-level coding benchmarks. SWE-bench (Jimenez et al., 2024) grades fixes for real GitHub issues with the fixing PR's tests, and its successors add languages, harder tasks, and live issues (Zan et al., 2025; Deng et al., 2026; Zhang et al., 2025). FEA-Bench (Li et al., 2025) and FeatureBench (Zhou et al., 2026) turn to feature development, which adds new capability, and Commit0 (Zhao et al., 2025) and ProgramBench (Yang et al., 2026) ask for whole libraries or programs. These benchmarks grade functional correctness with unit tests, mostly on CPUs, and do not measure performance, whereas SERVINGBENCH grades each patch on the released checkpoint and its target accelerator.

Performance benchmarks. SWE-fficiency (Ma et al., 2026), SWE-Perf (He et al., 2026), and GSO (Shetty et al., 2025) build repository-level tasks from performance-improving commits and compare the agent's speedup with the expert's, and ISO-Bench (Nangia et al., 2026) does so for vLLM and SGLang. KernelBench (Ouyang et al., 2025), JAXBench (Tschand et al., 2026), and FlashInfer-Bench (Xing et al., 2026) score single kernels against a reference, and InferenceBench (Yeon et al., 2026) deploys a fast server on one H100 for models that engines already support. In all of them, existing code already defines the correct behavior, whereas in SERVINGBENCH the agent must first make a new model run and match its outputs before its speed counts.

Serving-engine benchmarks. SWE-Serve (Williams et al., 2026) is the closest in domain: its 53 SGLang tasks span six families of inference engineering, some adding model or backend support, and three check performance, all on NVIDIA GPUs. TensorRT-LLM AutoDeploy (Liebenwein et al., 2026) automates part of model support by compiling PyTorch models into inference graphs. SERVINGBENCH is the only benchmark in Table 1 with all four marks: every task adds a model to a serving engine and is timed against the expert on NVIDIA and AMD GPUs or Google TPUs. Its reference patches are also the largest, editing 10.7 files and adding 1,485 lines, 45 functions, and 12 classes on average, 1.8× the lines of SWE-Serve.

3 SERVINGBENCH

Each SERVINGBENCH task comes from a *model bring-up PR*, a merged pull request that made a released model first servable in an engine. We call the engine just before the PR *Base* and the merged change *Gold*. The agent is provided with Base, the released checkpoint, and the target accelerator, and it submits a patch that is graded first on correctness and then on serving speed. The central difficulty is to check what Gold achieved without requiring the agent to reproduce how Gold achieved it, as many implementations can support the same model.

[2]FeatureBench is measured over its 166 L1 tasks, and KernelBench and InferenceBench provide no expert optimization patches for this comparison.

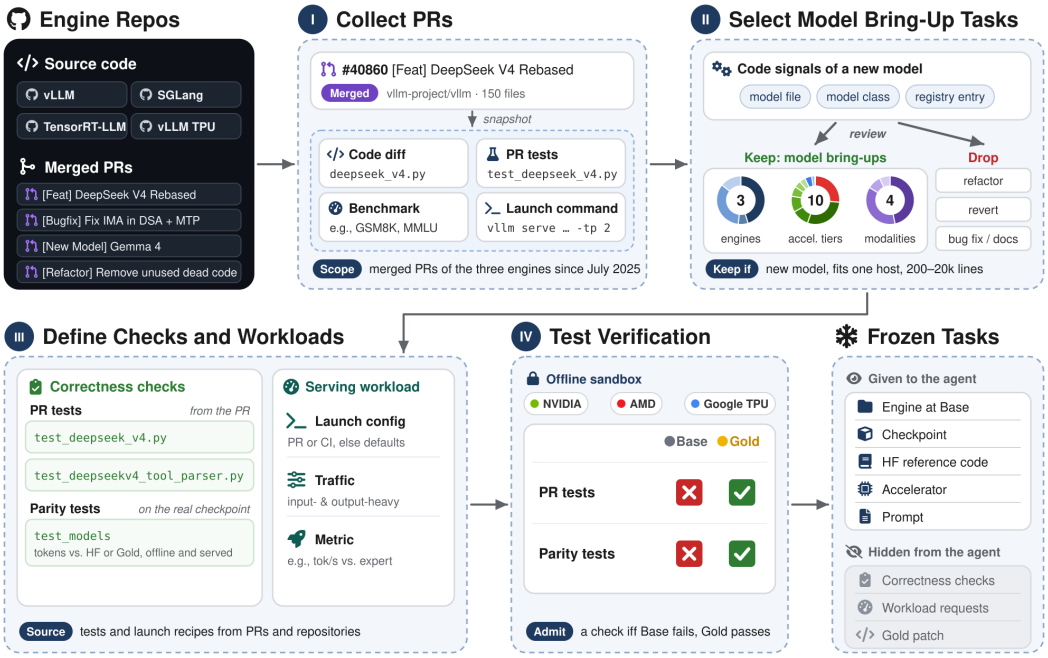


Figure 2: **Task construction pipeline.** Merged PRs from three serving engines (I) are filtered to model bring-ups (II), annotated with hidden correctness checks and a serving workload (III), and verified on the target accelerator, where a check is admitted only if Base fails it and Gold passes it (IV). The agent receives the engine at Base, the checkpoint, the Hugging Face reference code, the accelerator, and a prompt, while the checks, the workload requests, and Gold stay hidden.

Three principles guide benchmark construction (Figure 2). First, checks observe the engine solely through its public entry points and interfaces declared in the prompt, giving the agent freedom to structure its own implementation. Second, correctness and speed are measured using the released checkpoint on the target accelerator, as a model that successfully loads can still produce wrong outputs, and performance is hardware-dependent. Third, requirements are grounded in maintainer artifacts whenever available, with all remaining parameters held fixed across the benchmark.

3.1 Construction Pipeline

Stage I: Collect PRs. We collect pull requests of vLLM (including its TPU backend), SGLang, and TensorRT-LLM merged primarily between July 2025 and September 2026. For each PR, we keep its diff, description, tests, and benchmark or launch commands.

Stage II: Select model bring-up tasks. A bring-up makes a public checkpoint executable in an engine that could not run it before, unlike refactors, bug fixes, quantization-only updates, or ports of already supported architectures. Candidate PRs must add a new model file, model class, or registry entry, and we check each by hand against this definition. Selection does not constrain the time between a model's release and the merge, so a task replays a model's first bring-up rather than its release day. We filter for bring-ups whose checkpoint fits on one host and whose Gold patch modifies between 200 and 20,000 lines (excluding tests, documentation, examples, and benchmarks).

Stage III: Define checks and workloads. *Correctness checks.* Correctness is checked by two kinds of tests, parity tests and PR tests. The central check is an engine-level *parity test* that we write once per engine and instantiate for each task, so it names no symbol from the PR. It loads the released checkpoint through the engine's public API with the engine's native implementation and compares the outputs with Gold's by the rule of vLLM's own model tests (Appendix A.5). The same comparison also runs on the live server launched with the task's workload, so a patch must produce the right outputs when served as well. When the Hugging Face implementation runs on the target accelerator, a second parity test compares the engine with Hugging Face and so checks correctness

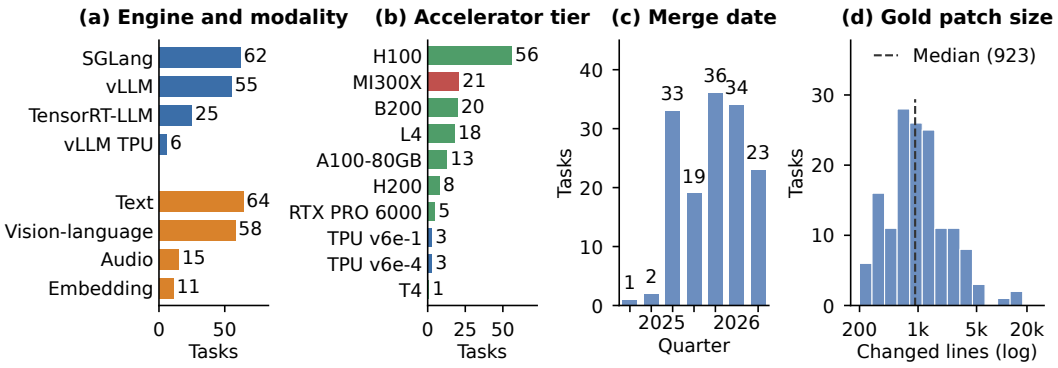


Figure 3: **SERVINGBENCH dataset.** Tasks per (a) serving engine (blue) and modality (orange), (b) accelerator tier by vendor (NVIDIA green, AMD red, Google blue), (c) PR merge quarter, and (d) Gold patch size in lines changed outside tests, documentation, examples, and benchmarks.

independently of Gold. On the other hand, the *PR tests* are the tests that the PR adds or modifies, except tests of Gold's internal kernels and helpers, which encode one implementation choice.

Serving workload. We follow the PR's performance setup whenever available, including its benchmark or workload and performance metric. Otherwise, we select the launch configuration from the engine's CI setup for the corresponding modality, then its documentation, and finally engine defaults, and use a standardized fallback workload. The fallback uses fixed random token sequences for text-generation models, with throughput measured at concurrency 32; fixed image/text inputs for vision-language models; and fixed audio clips or input batches for speech and embedding models. Requests are generated with a fixed seed by the engine's benchmark client pinned to its version at Base, so every implementation receives the same request stream. Each patch is timed in the same sandbox and with the same launch command as Gold. These measurements are designed for reproducible within-task comparisons rather than to model production traffic.

Stage IV: Test verification. Finally, Base and Gold run in a fresh sandbox on the target accelerator, graded by the same harness as agent patches. Following SWE-bench (Jimenez et al., 2024), a check is admitted only if Base fails it and Gold passes it. We also include pass-to-pass (P2P) tests, PR and CI tests that already pass on Base, and require Gold to pass them too. The Gold run also records the reference outputs of the parity tests. A candidate PR becomes a task only if at least one check is admitted and Gold passes every check and serves its workload. Otherwise we drop the candidate and record the reason with the failing run's log, such as a Gold that cannot load the released checkpoint.

Frozen tasks. The frozen task gives the agent the engine at Base, the released checkpoint, the target accelerators, and the Hugging Face model as a behavioral reference. The prompt states the support request, the capabilities that the task requires, the signature of each interface that a hidden test imports, and the workload shapes. The PR tests, parity tests, workload requests, and Gold stay hidden. The signatures are the minimum that a hidden test needs to call the patch, and, as in FEA-Bench (Li et al., 2025), the prompt reveals these interfaces but no part of Gold's implementation. To prevent shortcuts, the repository history ends at Base and the agent has no network access, so it cannot retrieve Gold or later upstream code. At grading, edits to test files are removed before the hidden tests are applied, and the patch cannot alter request generation or timing.

3.2 DATASET CHARACTERISTICS

SERVINGBENCH contains 148 tasks from 148 PRs, which add support for 125 released models (Figure 3). Three properties distinguish it from existing coding benchmarks.

Diverse engines, models, and accelerators. The tasks come from four repositories, each with its own model abstractions: SGLang (62), vLLM (55), TensorRT-LLM (25), and the TPU backend of vLLM (6). Besides 64 text models, they include 58 vision-language, 15 audio, and 11 embedding models, whose inputs take different paths through the serving engine. The tasks run on 10 accelerator tiers (Appendix B.3) from NVIDIA (121 tasks), AMD (21), and Google (6), so the benchmark is not tied to a single engine or accelerator.

Large changes across the serving engine. The size filter of Stage II keeps Gold patches of 200 to 20,000 changed lines, so no task is a one-line registration. Within this range, the median Gold patch changes 923 lines (Figure 3d). Measured as in Table 1, Gold adds 1,485 lines per task on average, 1.8× as many as SWE-Serve, including 45 new functions and 12 new classes. An agent must therefore write a new model and wire it into the serving engine, rather than edit a few existing lines.

Recent and extensible. Stage I collects PRs merged from July 2025 to September 2026, and every quarter in it contributes 19 to 36 tasks (Figure 3c). The same filters apply to new PRs, so the pipeline can add tasks as models are released.

3.3 Evaluation Metrics

Resolved rate. A patch *resolves* a task if it passes all of the task's correctness checks and serves the task's workload on the target accelerator. The *resolved rate* is the share of tasks resolved.

Performance and serving score. Each resolved patch P is compared with Gold, the maintainers' (human expert) implementation of the model. For performance, Gold is the merged PR. The *serving ratio* $\rho(P)$ compares the performance of P and Gold G under the same measurement setup. For a higher-is-better performance metric T, $\rho(P) = T(P)/T(G)$; for a lower-is-better metric, $\rho(P) = T(G)/T(P)$. When the PR does not specify a performance metric, T is aggregate throughput: output tokens per second for text-generation and vision-language models, and input throughput for speech and embedding models. A resolved patch is *slower*, *within 5%*, or *faster* when $\rho < 0.95$, $0.95 \leq \rho \leq 1.05$, or $\rho > 1.05$. As in SWE-fficiency, Gold is timed repeatedly, and its runs differ by a median of 0.8% (Appendix B.4). To combine correctness and speed, the *serving score* averages over all N tasks, giving 0 to an unresolved task:

$$S = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[P_i \text{ resolved}] \cdot \min\big(\rho(P_i), 1\big). \tag{1}$$

Normalizing by Gold puts tasks with different models, accelerators, and performance metrics on one scale. An unresolved task scores 0 because an incorrect bring-up cannot be deployed, and the cap at 1 keeps a speedup on one task from offsetting a failure on another.

4 Experiments

Setup. We evaluate seven state-of-the-art models (Claude Opus 5, GPT-5.6 Sol, Gemini 3.1 Pro Preview, Gemini 3.8 Flash, DeepSeek V4.1 Flash, Kimi K3, and GLM-5.3) in mini-SWE-agent (Lieret & Jimenez, 2025) with a budget of 200 steps and 4 hours per task (Appendix B.1). For each task, a model starts from the engine at Base on the task's accelerator, samples a single trajectory, and submits one patch (pass@1). We focus on pass@1 because a maintainer reviews one bring-up PR per model, while an agent can still build, test, and serve many candidate edits within one trajectory, as in SWE-bench and SWE-fficiency (Jimenez et al., 2024; Ma et al., 2026). Every run uses an isolated sandbox without network access (Table 9) on the task's accelerator (Appendix B.3).

4.1 Overall Results

Agents often fail to bring up the model. Claude Opus 5 is the strongest and fails only 22.3% of the tasks, while the other models fail 28.4% to 75.7% (Table 2). We use an LLM judge with Claude Opus 4.8 to categorize each failure by its cause (Figure 4a). Weaker models such as Gemini 3.8 Flash fail mostly at integration, because they do not understand the repository well enough to wire the model into the engine. For example, on openPangu-VL-7B in vLLM, Gemini 3.8 Flash imports a helper that the installed engine does not provide, so the model never loads.

On the other hand, for stronger models like Claude Opus 5, most failures come from a model that runs but computes the wrong outputs. For example, on Gemma-4-26B in TensorRT-LLM, it never reads the configuration fields for layers that share key-value caches, so the first generated token is already wrong. The remaining failures, mostly from GLM-5.3 and the two Gemini models, are code that does not build, crashes, or runs that end without a usable patch. Day-0 model support is far from

Model	Fails (%, ↓)	Serving score (↑)	Resolved, by speed (% of tasks) Slower (↓)	Within 5%	Faster (↑)	Slowdown (%) median / max
Claude Opus 5	22.3	0.72	25.0	36.5	16.2	18 / 46
DeepSeek V4.1 Flash	28.4	0.65	29.7	37.2	4.7	17 / 81
Kimi K3	29.7	0.65	27.0	35.1	8.1	15 / 87
GPT-5.6 Sol	39.9	0.54	18.9	32.4	8.8	25 / 97
GLM-5.3	48.0	0.48	15.5	30.4	6.1	17 / 78
Gemini 3.8 Flash	59.5	0.36	12.8	24.3	3.4	25 / 97
Gemini 3.1 Pro Preview	75.7	0.24	1.4	21.6	1.4	19 / 23
Average	43.3	0.52	18.6	31.1	6.9	18 / 97

Table 2: **Main results on SERVINGBENCH** (pass@1, 148 tasks). *Fails* is 100% minus the resolved rate (§3.3), and *Slower*, *Within 5%*, and *Faster* split the resolved tasks by serving speed against the human expert, so the four columns sum to 100%. *Slowdown* is the median and the largest loss in serving speed of the slower patches. The human expert has a serving score of 1.0, and the last row averages the seven models, except that its *Slowdown* pools their slower patches.

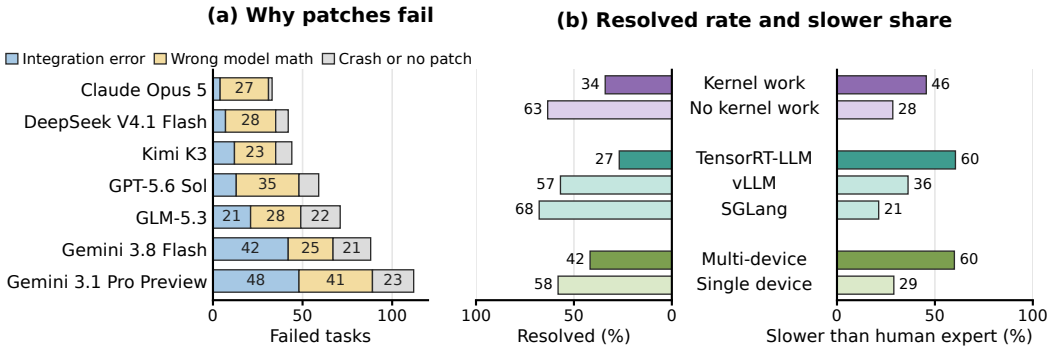


Figure 4: **Why patches fail, and where resolved patches are slower.** (a) Failed tasks per model, split into integration errors (model not registered or wired, wrong engine API, weights loaded incorrectly, or a missing kernel or backend), wrong model math (the model runs but computes different outputs), and crashes or no runnable patch. (b) Resolved rate (left) and share of resolved patches slower than the human expert (right), for tasks with and without kernel work (the human expert implementation writes or integrates a kernel), by serving engine, and by device count. Darker bars mark kernel work, TensorRT-LLM, and multi-device tasks; (b) averages the seven models.

solved: every model gets the model math wrong on a similar number of tasks, and stronger models mainly avoid integration errors and crashes.

Many resolved patches are slower than the human expert. Even for Claude Opus 5, 25.0% of the tasks are resolved with a slower patch, which is up to 46% slower than the human expert. About a third of slower patches lack an engine optimization that the human expert implementation has, and most of the rest use a slower layer kernel or attention backend. For example, on Qwen3-Next-80B in vLLM, Claude Opus 5 copies the key and value heads for every query head before the gated delta rule instead of calling the grouped kernel. On Kimi-Linear-48B in SGLang, it computes the MoE router in fp32 and routes through a Python function instead of the fused top-k kernel. Correctness alone does not make a bring-up fast.

Some open-weight models cost less but match the human expert less often. DeepSeek V4.1 Flash and Kimi K3 resolve nearly as many tasks as Claude Opus 5 at an API cost of $0.44 and $12.63 per task, under 2% and under half of its $27.72 (Figure 6a). On average, a run takes 0.7 to 2.4 hours depending on the model, so the models differ far more in API cost than in run time. However, their patches match or beat the human expert on about ten points fewer tasks (41.9% and 43.2% against 52.7%).

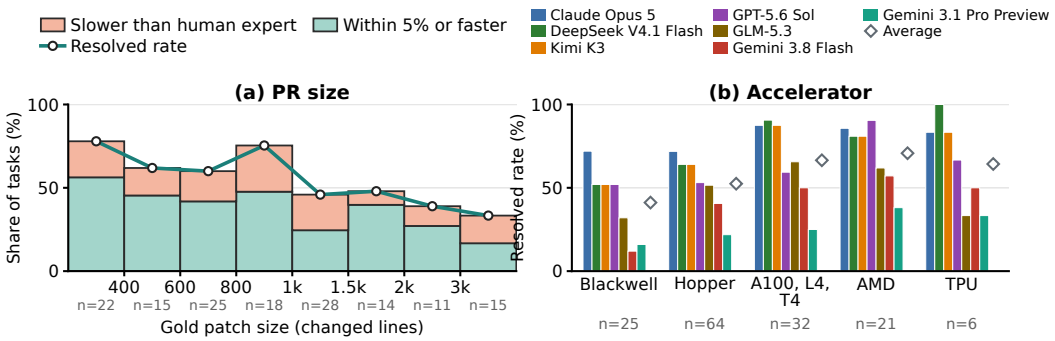


Figure 5: **Resolved rate by PR size and accelerator.** (a) Resolved rate (line) and resolved tasks slower than the human expert or within 5% or faster, by Gold patch size, with the task count of each bin below, averaged over models. (b) Per-model resolved rate by accelerator, with the average as a hollow diamond and task counts below. Blackwell: B200, RTX PRO 6000; Hopper: H100, H200.

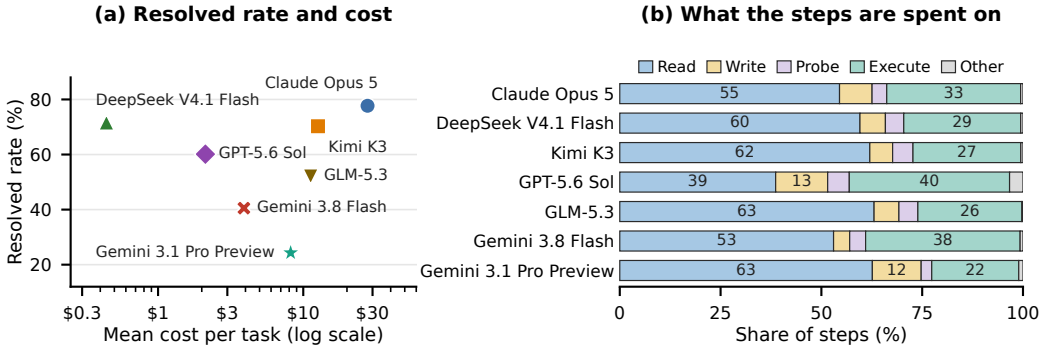


Figure 6: **Cost and how agents spend their steps.** (a) Resolved rate against mean model API cost per task at list prices, excluding accelerator time. (b) Share of each model's steps spent reading or searching code, writing code, probing the Hugging Face reference, executing builds, tests, or the serving engine, and other actions (process control, submitting, and reasoning-only turns).

4.2 Analysis

Unless a model is named, the numbers in this section average the seven models, each weighted equally.

Kernel work, serving engine, and device count. Figure 4b compares tasks by whether the human expert implementation writes or integrates a kernel, by serving engine, and by the number of devices. We observe lower resolved rates and more slower patches on tasks with kernel work (34% against 63%), in TensorRT-LLM (27% against 57% in vLLM and 68% in SGLang), and on multiple devices (42% against 58%). For example, on DeepSeek-V4-Flash in vLLM, the human expert implementation adds CUDA kernels for the new MoE routing, and no model writes them. TensorRT-LLM has about a quarter as many existing models to adapt, and for the same model family its tasks need kernel work 3.2× as often, so we hypothesize that agents there have less code to reuse. This suggests that agents fall furthest behind when a bring-up goes beyond the model file.

PR size and accelerator. Figure 5 explores how the resolved rate varies with PR size and accelerator. We observe a clear negative correlation between the resolved rate and PR size: PRs of 3,000 or more lines are resolved 2.3× less often than those under 400, suggesting that larger bring-ups remain challenging for current agents. For example, on Muse-Glimmer-30B in vLLM, the human expert implementation also adds a vision encoder and tool-call and reasoning parsers, and no model resolves the task. Every model resolves fewer Blackwell tasks than A100, L4, and T4 tasks (41.1% against 66.5% on average), and 10 of the 25 Blackwell tasks need kernel work. Holding kernel work fixed narrows the gap: Blackwell tasks without it are resolved 58% of the time, like Hopper (58%) but below A100, L4, and T4 (72%), and those with it 16% (against 33% and 34%).

Change	**Serving score**, default → with the change (difference)	
	Claude Opus 5	Gemini 3.8 Flash
Remove the Hugging Face implementation	$0.82 \rightarrow 0.62$ (-0.20)	$0.51 \rightarrow 0.39$ (-0.12)
Give another engine's patch	$0.82 \rightarrow 0.76$ (-0.06)	$0.51 \rightarrow 0.68$ ($+0.17$)
OpenHands instead of mini-SWE-agent	$0.82 \rightarrow 0.69$ (-0.13)	$0.51 \rightarrow 0.58$ ($+0.07$)

Table 3: **Ablations.** Serving scores with one change from the common defaults (Claude Opus 5: 0.82; Gemini 3.8 Flash: 0.51). Parentheses show differences.

Agents read far more than they run the model. Figure 6b splits each model's steps by the kind of action. All models but GPT-5.6 Sol spend most of their steps reading code, and a third to half of all runs never load the released checkpoint in the serving engine; for the three weakest models, runs that do load it resolve up to twice as often. Only Claude Opus 5 measures performance and runs tests in most of its runs.

Faster patches write their own kernels or pick a better configuration. Claude Opus 5, the one model that measures performance in most runs, is also faster than the human expert on 16.2% of the tasks, about twice as often as any other model (Table 2). Of all faster patches, 30% use a kernel that the agent writes, 20% a faster attention backend, and 23% an engine configuration that suits the workload better than the human expert's. For example, on Mamba-130M in SGLang, Claude Opus 5 writes a fused Triton kernel for the selective scan, which Gold computes in a PyTorch loop. Gold is thus not an upper bound for agents that do kernel or engine work.

4.3 Ablations

We evaluate three changes, one at a time, on 24 tasks balanced across serving engines, kernel work, and PR sizes, spanning NVIDIA GPUs (20), AMD MI300X GPUs (2), and Google TPU v6e (2).

Reference code helps both models. Removing the Hugging Face implementation lowers the serving score of Claude Opus 5 from 0.82 to 0.62 and that of Gemini 3.8 Flash from 0.51 to 0.39. With the reference, an agent can probe the released checkpoint's intermediate outputs against its own implementation and catch wrong model math, the most common failure of Claude Opus 5 (§4.1).

Another engine's patch helps only the weaker model. We give the agent another engine's Gold patch for the model. The patch raises the serving score of Gemini 3.8 Flash from 0.51 to 0.68 but lowers that of Claude Opus 5 from 0.82 to 0.76. Gemini 3.8 Flash fails mostly at integration (§4.1), and a working patch shows where a new model plugs into an engine.

Harness choice affects models differently. Replacing mini-SWE-agent with OpenHands lowers the serving score of Claude Opus 5 from 0.82 to 0.69 and raises that of Gemini 3.8 Flash from 0.51 to 0.58. In some Claude Opus 5 runs with OpenHands, long responses and repeated file writes truncated at the output limit exhaust the budget before the implementation is complete.

5 Conclusion

We introduced ServingBench, a benchmark of 148 day-0 model support tasks built from model bring-up PRs in three serving engines and verified on the released checkpoint and the target accelerator. Across seven state-of-the-art models, agents resolve 56.7% of tasks on average, and 33% of resolved patches are slower than the human expert. These results suggest that coding agents should run the released checkpoint and measure serving speed while they develop a patch. As serving engines integrate new model releases, the pipeline can keep adding tasks and extend ServingBench to more serving engines, accelerators, and modalities.

Reproducibility statement

Section 3 describes how tasks are collected, verified, and evaluated, and the appendix gives the serving workloads and additional results. Upon publication, we will release the benchmark tasks, their tests and serving workloads, and the evaluation code.

AI use statement

We used large language models to aid and polish the writing of this paper and to assist research execution: coding assistants helped write the scripts that construct, grade, and analyze the tasks and draw the figures, and an LLM judge categorizes failed patches (§4.1). The evaluated coding agents are the object of study and did not contribute to the paper. The authors designed the benchmark, checked all results and text, and take full responsibility for the content.

References

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/21bec6ace947b1b58967b945c8ac0f10-Abstract-Datasets_and_Benchmarks_Track.html.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R. Kundurthy, Sean M. Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? In *International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=uEVTdoAbnK.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. SWE-Perf: Can language models optimize code performance on real-world repositories? In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2507.12415.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/94074dd5a072d28ff75a76dabed43767-Abstract-Conference.html.

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? In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

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*, 2023. URL https://arxiv.org/abs/2309.06180.

Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. FEA-Bench: A benchmark for evaluating repository-level code generation for feature implementation. In *Annual Meeting of the Association for Computational Linguistics*, 2025. URL https://aclanthology.org/2025.acl-long.839/.

Lucas Liebenwein, Suyog Gupta, and Laikh Tewari. Automating inference optimizations with NVIDIA TensorRT LLM AutoDeploy. NVIDIA Technical Blog, 2026.

Kilian Lieret and Carlos E. Jimenez. mini-SWE-agent. https://github.com/SWE-agent/mini-swe-agent, 2025.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. SWE-fficiency: Can language models optimize real-world repositories on real workloads? In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2511.06090.

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. URL https://arxiv.org/abs/2602.19594.

NVIDIA. TensorRT-LLM. GitHub repository, 2026. URL https://github.com/NVIDIA/TensorRT-LLM.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *International Conference on Machine Learning*, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

SGLang Team. SGLang: A high-performance serving framework for large language models and multimodal models. GitHub repository, 2026. URL https://github.com/sgl-project/sglang.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-agents. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/ddd98389004161943e646322a16348c3-Abstract-Datasets_and_Benchmarks_Track.html.

Arya Tschand, Charles Hong, Julian Walker, Nina Cai, Shangkun Wang, Suvinay Subramanian, Sundar Dev, Vijay Janapa Reddi, Amir Yazdanbakhsh, and Sethu Sankaran. JAXBench: Benchmarking autonomous TPU kernel optimization. *arXiv preprint arXiv:2607.20466*, 2026. URL https://arxiv.org/abs/2607.20466.

vLLM Contributors. DeepSeek V4 rebased: Pull request 40860. GitHub pull request, 2026. URL https://github.com/vllm-project/vllm/pull/40860.

vLLM Team. tpu-inference: TPU inference for vLLM, with unified JAX and PyTorch support. GitHub repository, 2026a. URL https://github.com/vllm-project/tpu-inference.

vLLM Team. DeepSeek V4 in vLLM: Efficient long-context attention. vLLM Blog, 2026b. URL https://vllm.ai/blog/2026-04-24-deepseek-v4.

vLLM Team. Unit testing. vLLM documentation, contributing a new model, 2026c. URL https://github.com/vllm-project/vllm/blob/3b4566c5cf014605de6aeab6eb831b4f20511c17/docs/contributing/model/tests.md.

vLLM Team. Model test utilities (check_logprobs_close, check_embeddings_close). GitHub repository, tests/models/utils.py, 2026d. URL https://github.com/vllm-project/vllm/blob/3b4566c5cf014605de6aeab6eb831b4f20511c17/tests/models/utils.py.

vLLM Team. vLLM: A high-throughput and memory-efficient inference and serving engine for LLMs. GitHub repository, 2026e. URL https://github.com/vllm-project/vllm.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2407.16741.

Jennifer Williams, Dave Farris, Jeff Farris, and Jiantao Jiao. SWE-Serve: Benchmarking agentic engineering for production inference serving. *arXiv preprint arXiv:2609.26777*, 2026. URL https://arxiv.org/abs/2609.26777.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, 2020. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6/.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025. URL https://doi.org/10.1145/3715754.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie F. 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. In *Conference on Machine Learning and Systems*, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/37e44c4b5321605735be9761f9b758fc-Abstract-Conference.html.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

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. ProgramBench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026. URL https://arxiv.org/abs/2605.03546.

Jehyeok Yeon, Ben Rank, and Maksym Andriushchenko. InferenceBench: A benchmark for open-ended LLM inference optimization by AI agents. *arXiv preprint arXiv:2607.20468*, 2026. URL https://arxiv.org/abs/2607.20468.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/5afa9cb1e917b898ad418216dc726fbd-Abstract-Datasets_and_Benchmarks_Track.html.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. SWE-bench goes live! In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/d83c4a745789690f82e86d0ef752ae7c-Abstract-Datasets_and_Benchmarks_Track.html.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024. URL https://doi.org/10.1145/3650212.3680384.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. In *International Conference on Learning Representations*, 2025. URL

https://proceedings.iclr.cc/paper_files/paper/2025/hash/1fcefa894924bb1688041b7a26fb8aea-Abstract-Conference.html.

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. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2312.07104.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2602.10975.

A Dataset and Construction Details

This section describes where the tasks come from and under which licenses, how the 148 tasks break down by serving engine, how their reference patches compare with those of existing benchmarks, and what a task contains.

A.1 Repositories and Licenses

Table 4 lists the four repositories that the tasks come from. All four are public on GitHub under the Apache-2.0 license, which permits redistributing the Base revisions, Gold patches, and PR tests that the tasks contain. The tasks use 125 checkpoints released on the Hugging Face Hub under their own licenses: 63 under Apache-2.0, 24 under MIT, 30 under model-specific licenses such as the Gemma terms or the NVIDIA Open Model License, and 8 without a license on the model card. ServingBench does not redistribute checkpoints. Each task pins a checkpoint revision, which is downloaded from the Hub and staged before a run.

Repository	Description on GitHub	Tasks	License
`vllm-project/vllm`	"A high-throughput and memory-efficient inference and serving engine for LLMs" (vLLM Team, 2026e)	55	Apache-2.0
`sgl-project/sglang`	"SGLang is a high-performance serving framework for large language models and multimodal models." (SGLang Team, 2026)	62	Apache-2.0
`NVIDIA/TensorRT-LLM`	"TensorRT LLM provides users with an easy-to-use Python API to define Large Language Models (LLMs) and supports state-of-the-art optimizations to perform inference efficiently on NVIDIA GPUs." (NVIDIA, 2026)	25	Apache-2.0
`vllm-project/tpu-inference`	"TPU inference for vLLM, with unified JAX and PyTorch support." (vLLM Team, 2026a)	6	Apache-2.0

Table 4: **Source repositories** of the 148 tasks, with each repository's own description on GitHub, the number of tasks from it, and its license.

A.2 Dataset Statistics

Table 5 breaks the tasks down by serving engine. vLLM holds all AMD tasks, TensorRT-LLM tasks need kernel work most often and have the largest Gold patches, and the TPU backend has the largest share of multi-device tasks. *Checks per task* counts the graded correctness checks, each of which fails on Base and passes on Gold.

		Accelerator					Gold patch (mean)		
Engine	Tasks	NVIDIA	AMD	TPU	Text / Vision / Audio / Emb.	Kernel work / Multi-device	Files edited	Lines added	Checks per task
vLLM	55	34	21	–	15 / 23 / 7 / 10	22% / 7%	10.6	1,363	3.4
SGLang	62	62	–	–	35 / 21 / 6 / 0	11% / 6%	10.1	1,477	1.5
TensorRT-LLM	25	25	–	–	11 / 12 / 1 / 1	52% / 8%	13.6	1,923	1.8
vLLM TPU	6	–	–	6	3 / 2 / 1 / 0	33% / 50%	5.3	847	1.5
All	148	121	21	6	64 / 58 / 15 / 11	23% / 9%	10.7	1,485	2.2

Table 5: **Tasks by serving engine.** *Accelerator* counts tasks by vendor (Table 11). Modalities are text, vision-language, audio-language, and embedding or reranking models. *Kernel work* is the share of tasks whose expert writes or integrates a kernel, and *Multi-device* the share that runs on more than one GPU or TPU chip. Gold patch sizes are measured as in Table 1.

A.3 Comparison with Existing Benchmarks

Table 6 extends the last columns of Table 1 with more measures of how much code a reference solution writes. Every column is measured from the benchmark's released reference patches at a pinned revision with one extractor, so no number is copied from another paper. *Lines of oracle files* is the size of the files that the reference patch edits before the patch, and indicates how much existing code the agent has to understand. *Added components* are Python functions and classes that do not exist before the patch (the definition of FEA-Bench), and their share of the added lines indicates how much of a patch is new code rather than edits. ServingBench tasks add the most lines, functions, and classes, and only FeatureBench has a larger share of added lines in new components.

	SWE-bench						
	Full	Verified	FEA-Bench	SWE-fficiency	FeatureBench	SWE-Serve	ServingBench
Tasks	2,294	500	1,401	498	166	53	148
Repositories	12	12	83	9	24	1	4
# Lines of oracle files	1,960	1,487	2,749	4,215	14,218	10,131	7,314
# Files edited	1.7	1.2	3.4	2.2	14.4	10.1	10.7
# Lines added	27	10	123	32	759	828	1,485
# Lines edited	38	14	139	49	1,514	951	1,535
# Lines of added components	11	2	87	10	708	556	1,198
% Added components	41	21	71	31	93	67	81
# Functions added	0.73	0.25	4.49	0.90	29.90	25.43	45.41
# Classes added	0.06	0.01	0.78	0.16	0.12	4.40	11.92

Table 6: **Reference patch scale** of each benchmark, as means per task over its released reference patches. *Lines edited* counts added and removed lines. FeatureBench is measured over its 166 L1 tasks, the ones that release a patch. KernelBench and InferenceBench are left out because they release no reference patches.

A.4 Task Format

Table 7 lists the files of a task. The agent sees only the prompt, the repository at Base, the released checkpoint, and the Hugging Face reference code. Gold, the checks, Gold's reference outputs, and the serving workload stay hidden and are used only for grading.

File	Content	Agent sees
`task.json`	Engine, repository, PR, Base and merge commits, checkpoint and its pinned revision, accelerator tier and device count, and whether Gold writes or integrates a kernel (kernel work)	–
`prompt/PROMPT.md`	Task prompt: the model to support and what counts as supported, the interface to expose, how to load the checkpoint on the accelerator, the reference code, and how to check the patch	✓
`env/`	Environment of the engine at Base with its dependencies, built once per task and accelerator	✓
`gold/solution.patch`	Gold, the non-test changes of the merged PR	✗
`tests/tests.patch`, `tests/f2p.json`	Graded checks and the evidence that each fails on Base and passes on Gold	✗
`oracle/gold_outputs.json`	Gold's outputs on the fixed parity prompts	✗
`perf/workload.json`	Serving command and requests of the serving workload	✗

Table 7: **Files of a task.** The agent works in the environment at Base and receives the prompt, the checkpoint, and the Hugging Face reference code. The hidden files are used only for grading.

A.5 Correctness Contract

The parity tests follow the correctness tests that vLLM recommends for new models (vLLM Team, 2026c): `check_logprobs_close` for generative models and a cosine similarity for embedding models (vLLM Team, 2026d), with the token count, top-token count, and cosine threshold of vLLM's own model tests; for scores, class probabilities, and rewards we set a fixed tolerance band. A

generative model decodes three fixed prompts greedily for 32 tokens on the released checkpoint, and the tokens must agree up to their first difference, where each side's token must be among the other side's five most likely tokens. Table 8 gives the rule of the parity test against Gold for each output type; the parity test against the Hugging Face implementation applies `check_logprobs_close` or the cosine check through the engine's own model tests.

Output type	Compared output	Criterion
Generation	Greedy tokens of three fixed text prompts, 32 per prompt, with the five most likely tokens at each position	Equal up to the first difference, where each side's token is in the other's top 5; the first 8 tokens must be equal if an engine version reports no top tokens
Speech recognition (no text prompt)	The same, for three fixed synthetic 3-second audio clips	As for generation
Embedding	Mean-pooled embedding of each prompt	Cosine similarity ≥ 0.99
Scoring, classification, reward	Relevance scores, class probabilities, or rewards	$\lvert x - x_{\text{ref}}\rvert \leq 0.02 + 0.02 \max(\lvert x\rvert, \lvert x_{\text{ref}}\rvert)$ for every value

Table 8: **Parity rules** of the test against Gold, by output type (x_{ref} is Gold's value).

B Evaluation Details

B.1 Agent Configuration

Every model runs in the same harness with the same limits (Table 9).

Setting	Value
Harness	mini-SWE-agent (Lieret & Jimenez, 2025)
Step limit	200 steps
Wall-clock limit	4 h
Network access	None
Attempts per task	1 (pass@1)

Table 9: **Agent configuration**, shared by all models.

B.2 Prompt Template

Every prompt is rendered from the same template, whose sections are listed in Table 10. The template never includes implementation bodies from Gold, the hidden tests, their inputs, or the reference outputs.

Section	Content
Task	The PR title as the support request, and an instruction to implement it end to end in the repository.
Scope	The model architecture name from the checkpoint's configuration and the capabilities that the task requires, each derived from evidence such as the configuration (quantization, mixture-of-experts routing, multimodal inputs) or the serving setup (tensor parallelism), with the check that grades it.
Interface	The file path and signature of each interface that a hidden test imports, without bodies or docstrings, or the statement that no interface is prescribed.
Environment	The accelerator, the repository location, the local checkpoint with real weights, the editable install, and the absence of network access.
Reference	The local Hugging Face model as a behavioral reference, with the statement that it is not an implementation for the engine and exposes no grading inputs or outputs.
Self-checking	The statement that grading tests are held out, and a request that the agent write and run its own checks.
Performance	The workload shapes and metrics, the statement that throughput is measured without a threshold.

Table 10: **Sections of the prompt template.**

B.3 Hardware

Table 11 lists the accelerators of the 148 tasks, and Table 12 the machines that host them. All runs of a task, including Base, Gold, and every agent run, use the same accelerator and number of devices.

Accelerator	Memory per device	Devices	Tasks
NVIDIA H100 SXM	80 GB HBM3	1	56
NVIDIA B200	180 GB HBM3e	1–2	20
NVIDIA H200	141 GB HBM3e	1–4	8
NVIDIA A100	80 GB HBM2e	1	13
NVIDIA RTX PRO 6000	96 GB GDDR7	1	5
NVIDIA L4	24 GB GDDR6	1–2	18
NVIDIA T4	16 GB GDDR6	1	1
AMD Instinct MI300X	192 GB HBM3	1	21
Google TPU v6e-4	32 GB HBM per chip	4	3
Google TPU v6e-1	32 GB HBM per chip	1	3

Table 11: **Accelerators** of the 148 tasks. *Devices* is the number of GPUs or TPU chips a task uses. Each row is one accelerator tier. TPU v6e-4 and v6e-1 use the same chip and count as two tiers, one per Cloud TPU slice type, so the 10 tiers cover nine chip types.

Machine	Accelerators	CPU	Cores	Memory
H100 node	8× H100 SXM	2× Intel Xeon Platinum 8481C	104	1.8 TiB
B200 node	8× B200	2× Intel Xeon Platinum 8581C	112	3.8 TiB
Other NVIDIA GPUs	1–4 per run	–	8 / 16	256 / 128 GiB
MI300X host	1× MI300X	Intel Xeon Platinum 8568Y+	20	235 GiB
TPU v6e-1 VM	1 chip	AMD EPYC 9B14	44	176 GB
TPU v6e-4 VM	4 chips	AMD EPYC 9B14	180	720 GB
TPU v6e-8 VM	8 chips	2× AMD EPYC 9B14	180	1.4 TiB

Table 12: **Host machines.** *Cores* and *Memory* are per machine. A run on another NVIDIA GPU gets 8 cores and 64 to 256 GiB, and its grading 16 cores and 128 GiB. The agent's own timings never enter the score, and a patch and Gold are always timed with the same resources, together in one sandbox or each in a grading sandbox. MI300X runs get 16 cores. TPU v6e-4 and v6e-1 tasks also run on four or one chips of a v6e-8 VM.

B.4 Serving Measurement

Like SWE-fficiency, which times the expert and the agent repeatedly in the same environment (Ma et al., 2026), we measure Gold more than once. A timing sandbox on the task's accelerator serves Gold, then each resolved patch of the task, then Gold again, with the same launch command and request stream. Before each timed configuration, one untimed pass with a different seed warms the server, and the prefix and multimodal caches are flushed. A patch's serving ratio divides its speed by the mean of the two Gold runs. Across 780 timing sandboxes on 128 tasks, the two Gold runs differ by a median of 0.8% and by 5.4% at the 90th percentile, and 11% of sandboxes differ by more than 5%. The 5% band is thus well above the typical Gold-to-Gold difference, but a patch near 0.95 or 1.05 can change bucket on a repeat.

C Additional Results

C.1 Another Engine's Patch

In the treated run, the other engine's Gold patch is placed read-only in the sandbox, and one added prompt section points to it and asks the agent to follow the target repository's own conventions; everything else matches the control, the model's run in the main evaluation. The patch helps Gemini 3.8 Flash, which fails mostly at integration, find where a new model plugs into the engine. For Claude Opus 5, which fails mostly at model math rather than integration, the patch likely misled it on the 2 tasks it loses: it follows the other engine's code where the two engines structure the model differently, and its outputs no longer match Gold's.

C.2 Steps and Time per Action

Figure 7 adds the share of wall-clock time to the step shares of Figure 6b. Reading takes 39% to 63% of each model's steps but at most 17% of its time; running builds, tests, and the serving engine

takes the largest share of time for five of the seven models, and waiting for the model's responses for Kimi K3 and Gemini 3.8 Flash.

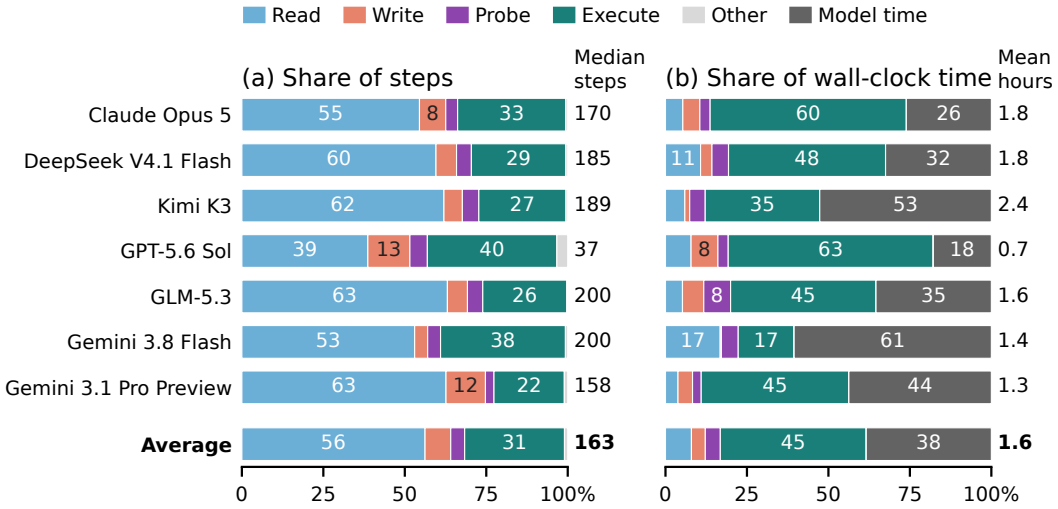


Figure 7: **Share of steps and of wall-clock time per action type**, with the median steps and the mean hours per run over all tasks (as in §4.1). Shares are over the runs whose step-level trajectory is available, reweighted to each model's resolved rate. Probe runs the Hugging Face reference model, execute runs scripts, tests, builds, and the patched model in the engine, and model time is spent waiting for the model's next step.

Figure 8 shows how the mix changes over a run: most models read first and run code more toward the end. Resolved runs spend 4.8% of their steps running the Hugging Face reference model against 3.6% for unresolved runs, and in the last tenth of a run, unresolved runs spend 17% of their steps writing code against 12% for resolved runs.

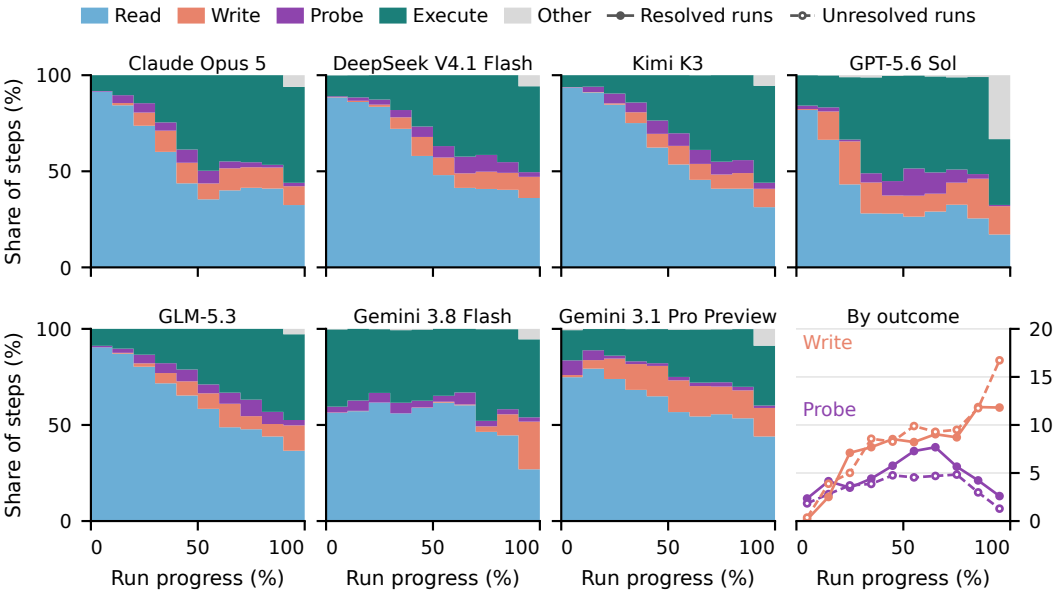


Figure 8: **Action mix over the course of a run** for each model, with every run cut into ten equal parts by step. The last panel shows the write and probe shares of resolved and unresolved runs, averaged over the models.

C.3 Resources per Task

Table 13 lists the resources behind Figure 6a. DeepSeek V4.1 Flash and Claude Opus 5 read about as many input tokens per run (27.6M and 28.1M), so the 63× difference in their API cost comes from the price per token, not from longer runs. The main agent runs used about 2,000 accelerator-hours (devices times wall-clock) and their grading about 300 more; validating Gold took about 400 and the serving measurements about 330.

Model	Steps	Agent hours	Input tokens (M)	Cost
Claude Opus 5	170	1.81	28.1	$27.72
DeepSeek V4.1 Flash	185	1.83	27.6	$0.44
Kimi K3	189	2.44	23.0	$12.63
GPT-5.6 Sol	37	0.72	2.9	$2.12
GLM-5.3	200	1.64	21.4	$11.25
Gemini 3.8 Flash	200	1.39	23.2	$3.90
Gemini 3.1 Pro Preview	158	1.35	11.1	$8.19

Table 13: **Resources per task on the main runs**: the median steps, the mean agent hours and input tokens, and the mean model API cost per task at list prices (OpenRouter prices, and OpenAI's for GPT-5.6 Sol), with cached input at the cache-read price and reasoning tokens as output. The cost does not include accelerator time.

D Limitations

ServingBench evaluates model bring-up across three serving engines and 10 accelerator tiers, with each task running on at most four devices of one host. Extending the benchmark to multi-node serving would test the additional communication and coordination required to support models that exceed a single host's capacity.

The tasks reconstruct a model's first integration from a historical pull request. The engine is pinned at Base, while the checkpoint and reference code are pinned to a Hub revision from around the merge. When needed, Transformers is upgraded to the earliest release that loads the checkpoint. These environments therefore evaluate first-time model support with reference material that may have become available after the model's release. Evaluating release-day conditions would additionally require restricting that material to the versions available on the release date.

Each environment is built at Base and also installs the packages that Gold adds, since the sandbox has no network access, and Gold is validated in the same image, so the image holds every package that Gold needs; nine Gold patches change a dependency or build file, and none upgrades a kernel library. The TPU backend runs on an installed vLLM pinned to Base's date, which already implements the architectures of all 6 TPU tasks in PyTorch, and three Gold patches run that implementation through torchax. An agent on these tasks can read that implementation, and checks that Base already passes through it are dropped.

Public repositories make these tasks reproducible, but their merged solutions may also appear in the training data of the evaluated models, as with other repository-level benchmarks (Jimenez et al., 2024; Zhou et al., 2026). The extent of this overlap is difficult to establish without access to the models' training corpora. We keep Gold hidden, truncate repository history at Base, and disable network access during evaluation. These measures prevent agents from retrieving the solution during a run, but prior exposure during training remains possible. The construction pipeline supports adding newly merged bring-ups, allowing future evaluations to select tasks whose solutions were first made public after a model's training cutoff. Continually adding such tasks follows the approach of LiveCodeBench (Jain et al., 2025) and SWE-rebench (Badertdinov et al., 2025).