INFRADEV: CAN CODING AGENTS DEVELOP AI INFRASTRUCTURE, AND HOW CAN WE TELL?

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are beginning to develop the production-grade AI infrastructure that trains and serves large models, yet existing benchmarks cover only single kernels or single-GPU tasks, and grade with tests and prompts that misjudge both correct and incorrect patches. We introduce InfraDev, a benchmark of 36 tasks built from real-world pull requests to Megatron-LM, vLLM, and verl, covering infrastructure development for training, inference, and reinforcement learning. Each task asks an agent to reimplement a real change that spans many files and must run correctly across many GPUs. To grade such changes, we jointly synthesize hidden probes, which enforce the requirements the original tests leave unchecked, and a prompt that states every interface the probes call without revealing the solution. We also design a shared execution environment that reuses dependencies and artifacts across tasks, cutting environment build time by 30–50% and storage by 42–46% on Megatron-LM and vLLM. Across six frontier models, the best resolves only 38.9% of the tasks; the probes reject 32% of the submissions that pass every official test, and with the original pull request descriptions as prompts, no model resolves a single task. Failure analysis shows that structural and state errors dominate and that agents' own tests rarely check memory, time, or traffic. Turning these findings into an agent skill raises the resolved rate of a frontier model on the Megatron-LM tasks by 10.5%.

1 INTRODUCTION

Real-world AI infrastructure spans training, inference, and reinforcement learning (Shoeybi et al., 2020; Kwon et al., 2023; Zheng et al., 2024; Sheng et al., 2025). Each is backed by a large codebase that evolves rapidly with new models and hardware: vLLM alone merged 1,918 commits in June 2026, about 64 per day (Luu, 2026). Developing this infrastructure means implementing changes that range from a single GPU kernel to coordinated edits across parallelism, memory, and communication subsystems. Such a change must be correct in two senses. It must be *numerically correct*, computing the right activations, gradients, and states under distributed execution, and it must be *behaviorally correct*: a new feature must take effect, and an optimization must actually reduce memory, communication, or time on realistic workloads. Coding agents are beginning to make these changes, as they take on longer and more complex software engineering tasks (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025b; Deng et al., 2025; Kwa et al., 2025). This raises the question: **Can coding agents develop real-world AI infrastructure, and how can we tell?**

Answering this question requires benchmarks that address three limitations of existing evaluations. First, existing benchmarks have a **narrow task scope**. Most evaluate individual GPU kernels (Ouyang et al., 2025; Li et al., 2025; Chan et al., 2026; Huang et al., 2026) or single-GPU inference optimization (Nangia et al., 2026; Yeon et al., 2026). Φ-Bench recently extended the scope to system-level optimization (Ding et al., 2026), but its tasks are optimization problems with fixed targets rather than the changes developers actually ship. Most real infrastructure changes go further: they span many files and modules and must run correctly across many GPUs. Second, reusing a pull request's own tests and description as the grading suite and task prompt (Jimenez et al., 2024) yields **unreliable grading** in both directions. The tests are incomplete on both counts. Among the 36 pull requests we study, 24 have no test that checks a numerical result, and 13 of the 15 performance optimizations have no test that checks the promised saving, so a patch that skips the

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.

optimization still passes. The prompts are underspecified. They omit interface names and behaviors that the tests depend on, so a correct implementation can fail. The two problems are coupled, because every requirement a test enforces must also be stated in the prompt. Prior work strengthens tests (Yu et al., 2025; 2026) or enriches prompts (Deng et al., 2025), but not both together. Third, evaluation on real infrastructure has **high cost**. Each task pins its own dependencies and needs large artifacts such as model weights. Preparing them separately for every task takes substantial time and storage, which limits how many tasks a benchmark can afford and who can run it.

We address these limitations with three contributions, and then use the resulting benchmark to understand why agents fail and to improve them.

A benchmark of large-scale AI infrastructure tasks. InfraDev contains 36 carefully selected tasks built from merged pull requests to Megatron-LM, vLLM, and verl, covering training, inference, and reinforcement learning. We start from the 21,389 pull requests merged into these repositories between January 2025 and September 2026. We keep those that change core functionality, touch multiple files, add tests, and exceed 1,000 lines. Each remaining task asks an agent to reimplement one pull request. A task changes 17.4 files and 1,818 lines on average, and 20 of the 36 require multiple GPUs to run.

A method that builds tests and prompts together. We propose hidden-probe synthesis, implemented as an agent. For each selected pull request, the agent first writes adversarial patches that pass the official tests but violate requirements the tests leave unchecked. It then writes new tests for these requirements, which we call *hidden probes*. A probe is kept only if it passes an acceptance check: it must accept the golden patch in repeated runs and reject the pre-PR snapshot, every adversarial patch, and a hollow patch that defines the new interfaces without implementing them. Finally, the agent writes a prompt that states every behavior the probes check and every interface they call, without revealing how to implement them. Both halves are necessary: the probes reject 32% of the submissions that pass all official tests, and with the original pull request descriptions as prompts, no model resolves a single task.

A resource-efficient execution environment. We design an execution environment in which tasks with compatible dependencies share one environment, built once and keyed by a hash of its specification. Model weights and datasets are downloaded once and shared across tasks. This cuts environment build time by 30–50% and storage by 42–46% on Megatron-LM and vLLM, compared with building one image per task as prior benchmarks do (Nangia et al., 2026; Yeon et al., 2026).

Using InfraDev, we evaluate six frontier coding agents. The best, GPT-6 Sol, resolves only 38.9% of the tasks (Section 4.2). We then analyze the 1,754 failing tests of the graded submissions and find failure modes that agents repeat: structural errors are the most common, state errors often escape the official tests, the tests agents write for themselves rarely check memory, time, or traffic, and two thirds of the errors are shared by most models.

Finally, we encode these findings as **an agent skill for AI infrastructure development**. The skill summarizes the failure cases and incorporate several development principles revealed by InfraDev. Our experiments show that this skill raise the resolved rate of GPT-5.6 Sol on the Megatron-LM tasks from 42.1% to 52.6%, where the failure modes it targets are most common (Section 4.5).

Encouraged by the above results, we will open-source InfraDev and believe it offers a foundation for measuring and improving coding agents on large-scale AI infrastructure.

2 Related Work

Benchmarks for AI infrastructure and ML engineering. Most benchmarks in this area evaluate GPU kernel generation and optimization (Ouyang et al., 2025; Li et al., 2025; Xing et al., 2026; Lange et al., 2025), extended to multiple GPUs by ParallelKernelBench (Chan et al., 2026) and to kernels inside their host frameworks by RealisticTritonBench (Huang et al., 2026). ISO-Bench, InferenceBench, and Φ-Bench move up to inference optimization, but on a single GPU and with optimization targets fixed in advance (Nangia et al., 2026; Yeon et al., 2026; Ding et al., 2026). A separate line evaluates agents on ML engineering, such as training models for Kaggle competitions or reproducing papers (Chan et al., 2025; Starace et al., 2025; Wijk et al., 2025); there the agent uses infrastructure rather than changing it. Despite similar names, InfraBench targets cloud and

operations agents (Gao et al., 2026) and SWE-InfraBench targets infrastructure-as-code (Tarasova et al., 2026). InfraDev differs from all of these in its unit of work: a merged pull request to a production training, inference, or RL framework, which typically changes many files and must run correctly across GPUs.

Grading in repository-level benchmarks. SWE-bench and its descendants grade a patch with the tests that shipped with the original pull request (Jimenez et al., 2024; Zan et al., 2025; Deng et al., 2025), and infrastructure benchmarks extend them only with end-to-end checks of accuracy and speed (Nangia et al., 2026; Huang et al., 2026). Both weaknesses of this recipe are known. On the test side, UTBoost adds tests to raise coverage (Yu et al., 2025), SWE-ABS uses adversarial patches to show that many accepted patches are wrong (Yu et al., 2026), and work on verifiable rewards reports policies that learn to pass weak tests without solving the task (Baker et al., 2025). On the prompt side, SWE-bench Verified removes underspecified tasks by human annotation (Chowdhury et al., 2024) and SWE-Bench Pro adds requirement and interface lists (Deng et al., 2025). Each fixes one side. InfraDev fixes both in one loop: adversarial patches find requirements the official tests miss, hidden probes enforce them, and the prompt is rewritten to state every interface the probes depend on (§3.1.2). Environment reuse is a smaller but practical concern: prior benchmarks build one image per task (Jimenez et al., 2024; Nangia et al., 2026) or share only a base image (Huang et al., 2026); we share dependency environments and artifact caches across tasks (§3.3).

Understanding and improving coding agents. Failure analyses of agents on SWE-bench and related benchmarks report where runs break down, and agents are improved by better interfaces (Yang et al., 2024), reusable skills and tool libraries, or reflection over past attempts. Kernel-generation systems apply the same ideas to one task type (Lange et al., 2025; Wang et al., 2025a; Baronio et al., 2026). Our skill differs in where it comes from: it encodes the three failure classes, contains no task-specific answers, and targets the checks that agents skip, such as callers of a changed function and the resource a change promises to save.

3 INFRADEV

Each task in InfraDev asks a coding agent to reimplement a real merged PR (pull request) from an AI infrastructure project. We first curate challenging PRs that change core functionality of Megatron-LM, verl, and vLLM (§3.1). For each PR, we jointly synthesize the tests and the prompt of the task. An agent adds hidden probes that reject shortcut implementations of the PR. Another agent writes a prompt that states what these tests check without revealing the solution (§3.1.2). The agent under evaluation then implements the task in a multi-GPU sandbox, based on the prompt and the repository snapshot just before the PR. We grade its patch with the *official tests* of the PR and the *hidden probes* (§3.2). All stages run in a reusable execution environment (§3.3) for accleration.

3.1 TASK CONSTRUCTION

3.1.1 PR CURATION

We collect tasks from three widely used production repositories that together cover the model life cycle: Megatron-LM for training (Shoeybi et al., 2020), verl for reinforcement learning (Sheng et al., 2025), and vLLM for inference (Kwon et al., 2023). We apply rule-based filters to all of their PRs and keep those that are merged and not reverted, change at least 1,000 lines, and add new tests. Guided by domain expertise, we then manually select the PRs that concern the core functionality of each framework. This yields 36 PRs, most of which add new features or optimize performance (Table 1). For each PR, we record the pre-PR snapshot just before the merge and the post-PR snapshot right after it. Their difference is the *golden patch*, the change the developers actually made, which serves as the reference solution.

3.1.2 JOINT SYNTHESIS OF TESTS AND PROMPT

Hidden-probe synthesis. We judge whether a modification completes a task by two criteria. *Behavioral correctness* means that the program behaves as the task requires when it runs: a new feature

Table 1: Task types of the 36 selected pull requests. *New feature*: adds functionality that the pre-PR snapshot lacks; *Performance*: keeps the functional result but reduces time, memory, or communication; *Refactor*: restructures code without adding functionality; *Bug fix*: corrects wrong behavior.

Repository	New feature	Performance	Refactor	Bug fix	Total
Megatron-LM (training)	9	8	1	1	19
verl (post-training)	5	2	0	0	7
vLLM (inference)	5	5	0	0	10
Total	19	15	1	1	36

Algorithm 1 Hidden-probe synthesis.

Require: PR r, its golden patch Δ_{gold}, the pre-PR snapshot $\mathcal{S}$, and the official tests T_{off}
Ensure: hidden probes T_{probe}
1: $C \leftarrow \textsc{Summarize}(r)$ ▷ what r promises to implement
2: $\Delta_1, \ldots, \Delta_k \leftarrow \textsc{BuildAdversarialPatches}(C, T_{\text{off}})$ ▷ wrong patches that pass T_{off}
3: $T_{\text{probe}} \leftarrow \textsc{WriteProbes}(C)$ ▷ check both criteria
4: **while** not $\textsc{CheckProbes}(T_{\text{probe}}, \Delta_1, \ldots, \Delta_k)$ **do**
5: Revise T_{probe} and $\Delta_1, \ldots, \Delta_k$ to fix the failed checks
6: **return** T_{probe}

must take effect, and an optimization must actually reduce memory, communication, or time. *Numerical correctness* means that the program computes the right values, such as activations, gradients, and model parameters.

The official tests of the selected PRs are weak in both criteria. First, numerical checks are often absent: in 24 of the 36 evaluated tasks, no official assertion verifies the numerical result of the modification. Second, behavioral checks are rare: in 13 of the 15 PRs that optimize performance, the official tests never check whether the optimization saves time, memory, or communication. An implementation that computes the right output but omits the optimization therefore still passes them.

We address this gap with an agent skill that iteratively synthesizes *hidden probes*, tests for requirements that a task imposes but its official tests leave unchecked (Algorithm 1). The agent first summarizes what the PR promises to implement (line 1). It then builds *adversarial patches*, incorrect implementations that pass every official test but violate numerical or behavioral correctness (line 2). Next, the agent writes probes that check both criteria (line 3). Appendix A.1 describes how a probe decides whether a result is correct. For example, verl PR 6974 syncs weights from the trainer to the rollout engine by sending only the entries that changed since the last step. Its probe counts the sent entries and rejects an adversarial patch that sends 65,536 entries when only 655 have changed.

We then refine the probes and the adversarial patches together (lines 4–5). In each round, the skill runs a check, and the agent revises every probe or adversarial patch that fails. The check requires each probe to pass on the golden patch and fail on every adversarial patch. It also requires each adversarial patch to break a promise of the PR while still passing the official tests. Appendix A.1 lists the full check.

Prompt synthesis. To improve the quality of prompts, we rewrite each PR as a task prompt, the agent's only description of the change. The prompt must be *sufficient*: a correct implementation must not fail because the prompt omits an interface or requirement on which the tests depend, such as the name of a function they import. It must also be *solution-free*: it must not explain how to meet the requirements, because such guidance does part of the task and makes weaker agents appear capable. We meet both requirements by splitting the information about a PR into a *specification* and a *solution* and putting only the specification in the prompt. The specification states *what* the change must do: the interfaces the tests call, the behaviors the official tests and hidden probes check, and the execution rules, such as the definition of done. The solution describes *how* to do it: algorithms, formulas, numerical constants and tolerances, and validation strategies. We omit any item that could belong to either.

A second agent skill constructs the prompt. It receives from probe synthesis a handoff record that lists the claimed behaviors and the required symbols with input types, but not the probe code. A

static analyzer first extracts the *interface contract*: the symbols that the golden patch introduces and that the official tests or the hidden probes reference. An authoring agent then writes the prompt from the PR description, the golden patch, the official tests, the handoff record, and the contract. It states each claimed behavior either as a requirement or as a decision explicitly left to the agent. Finally, the prompt must pass two checks: a linter rejects any prompt that omits a contract symbol, and a checker agent rejects any prompt that discloses part of the solution. The author revises the prompt until it passes both checks.

3.2 Evaluation

Inference and Grading. We prepare Docker containers for all agents and set a 2.5 hours time limit for each PR. Recovery mechanism is also introduced to prevent agent stopped accidentally in early stage. Agent's output patch is applied to the pre-PR snapshot in a separate container. The test directory is then replaced by that of the post-PR snapshot for evaluation, so agents cannot cheat by modifying test. To quantitatively evaluate how good agents solve the task, grading will evaluate patches with real GPUs.

Metrics. We grade each patch on the official tests together with the hidden probes. For a task p, let N_p be the number of test cases collected when the golden patch runs, including the cases that the golden patch itself fails or skips on our hardware. The pass rate of a patch is $s_p = \text{passed}_p / N_p$, and % Passed averages s_p equally over tasks. A task is resolved when the patch passes at least as many test cases as the golden patch, so the patch does not have to pass the cases that the golden patch fails. % Resolved is the share of resolved tasks. A missing, empty, or inapplicable patch scores $0/N_p$ instead of being dropped, since dropping it would reward models that give up. Harness faults are excluded and counted separately, so an infrastructure error is never recorded as a model failure. We keep the per-test outcomes and the full agent trace of every run for the failure analysis in §4.4.

3.3 Execution Environment

Limitation of Docker. Repository-level benchmarks usually build one Docker image per task. Docker caches image builds as a prefix, reusing the build steps before the first changed step and rerunning every step after it. This prefix cache works poorly for AI infrastructure projects, whose dependencies change often. For example, the 19 Megatron-LM tasks were merged within five months but use six versions of Transformer Engine. Each version change makes Docker rerun dependency steps after it. Moreover, tests of production infrastructure need large artifacts, such as model weights and datasets. The test code downloads these artifacts while it runs, so Docker does not see them and cannot cache them across tasks.

Resource-efficient Execution Environment. We break the prefix reuse of Docker and reuse each package and each artifact independently, so a change to one of them never forces the others to be rebuilt or downloaded again. We maintain several lightweight base Docker images for each AI infrastructure repository, containing only components shared across multiple PRs, such as CUDA and the Python interpreter. The dependencies of each task go into a virtual environment that `uv` installs on a volume shared by all containers. Tasks with the same hash of their locked dependencies share one environment, and a new environment hard-links cached packages instead of reinstalling them. Every container also mounts one shared Hugging Face cache, so each model or dataset is downloaded only once. Both agent inference and grading build their environments this way.

4 Experiment

4.1 Setup

Models and harnesses. We evaluate six coding agents on InfraDev, two built on frontier closed-source models and four on open-source models. GPT-6 Sol and GPT-5.6 Sol run in OpenAI's Codex harness; GLM 5.3, DeepSeek V4 Pro (0813), Kimi K3, and HY4 preview run in the OpenCode harness.[1] We set the reasoning effort of every model to `xhigh` or a comparable level. Each agent runs once on each task, and we call the patch it produces a *submission*.

[1] We use Codex version 0.155.1 and OpenCode version 1.18.31.

Table 2: Results of 6 models on InfraDev. % Resolved is the share of tasks on which every test passes; % Passed is the share of tests passed on a task, averaged over tasks. Time, Cost, and Tokens are per-task means of the agent's run time, API cost estimated at list prices, and tokens consumed.

Model	% Resolved	% Passed	Time (min)	Cost ($)	Tokens (M)
GPT-6 Sol	**38.9**	**78.8**	**43.9**	5.37	**20.86**
GPT-5.6 Sol[2]	30.6	78.6	**73.9**	17.98	**24.71**
GLM 5.3	25.0	76.7	136.9	13.89	48.84
HY4 preview	**36.1**	**83.3**	118.1	**2.09**	31.75
DeepSeek V4 Pro	30.6	74.3	94.1	**1.89**	41.80
Kimi K3	30.6	75.8	136.4	13.48	33.58

Table 3: % Resolved per task category. The number of tasks is in parentheses.

Model	Training (19)	Inference (10)	RL (7)
GPT-6 Sol	**52.6**	10.0	**42.9**
GPT-5.6 Sol	**42.1**	10.0	28.6
GLM 5.3	26.3	10.0	**42.9**
HY4 preview	**42.1**	10.0	**57.1**
DeepSeek V4 Pro	**42.1**	10.0	28.6
Kimi K3	31.6	10.0	**57.1**

Environment. All experiments run on nodes with eight NVIDIA H800 or H20 GPUs each. Each inference or grading run has exclusive use of one node. An agent has 2.5 hours per task, excluding environment preparation; a run that reaches this limit is graded on the patch it has produced. A task's golden calibration and the grading of its submissions use the same environment and GPU type.

4.2 Results on InfraDev

Table 2 shows that InfraDev remains challenging for all models: the resolved rate ranges from 25.0% for GLM 5.3 to 38.9% for GPT-6 Sol. Across task categories (Table 3), inference tasks are the hardest: every model resolves only one of the ten. Resolved rates spread more widely on training tasks (26.3% to 52.6%) and on RL tasks (28.6% to 57.1%). No single model leads in all categories: GPT-6 Sol performs best on training tasks, and HY4 preview and Kimi K3 tie for best on RL tasks.

A high passed rate does not mean that a model completes tasks. Every model passes between 74.3% and 83.3% of the tests, but none resolves more than 38.9% of the tasks, because a task is resolved only when all its tests pass. Nearly equal passed rates can also hide different resolved rates: GPT-6 Sol and GPT-5.6 Sol pass 78.8% and 78.6% of the tests but resolve 38.9% and 30.6% of the tasks. The models differ on a few precise tests, which a patch that only looks complete cannot pass.

No model we test leads in resolved rate, time, and cost at once. GPT-6 Sol comes closest: it has the highest resolved rate, is the fastest at 43.9 minutes per task, and costs less than GPT-5.6 Sol, GLM 5.3, and Kimi K3. HY4 preview resolves one task fewer at 39% of the cost but takes 2.7× as long. Cost and time also depend on the provider's prices, caching, and serving speed, not only on the model.

4.3 Ablation Study

Hidden-probe synthesis. The official tests alone overstate how many tasks agents resolve. We grade every submission twice, on the official tests alone (Set A) and with the hidden probes added (Set B; Figure 1). Adding the hidden probes lowers every model's resolved rate by 11 to 19 percentage points. Of the 216 submissions, 102 pass all official tests, but only 69 of them also pass the hidden probes. The hidden probes thus reject 32% of the submissions that the official tests accept.

[2]GPT-5.6 Sol was measured through the Codex coding plan between September 21 and 24, 2026; we observed possible degradation of the served model around these dates.

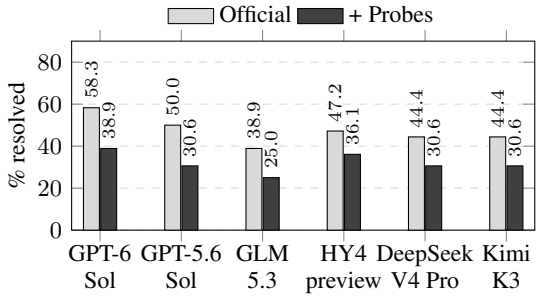


Figure 1: Resolved rate (%) on the 36 tasks, with and without the hidden probes.

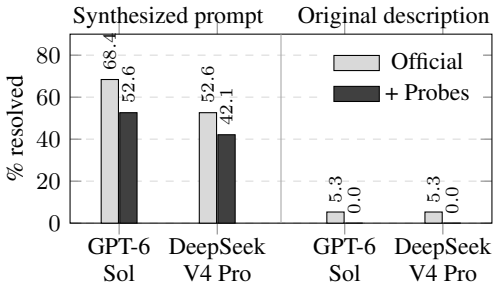
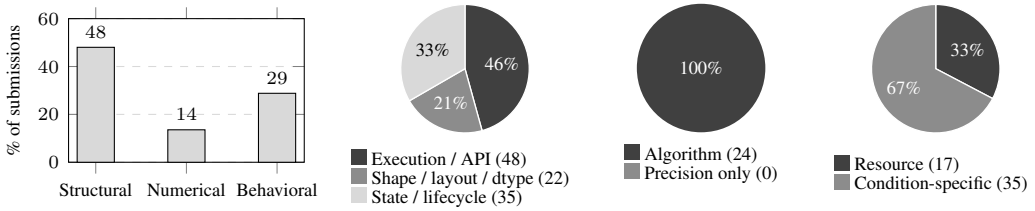


Figure 2: Resolved rate (%) on the 19 Megatron-LM tasks. *Original description* prompts the agent with the PR's own description on GitHub.



(a) Error class (b) Structural Errors (c) Numerical Errors (d) Behavioral Errors

Figure 3: (a) Share of the graded submissions with at least one error of each class. (b)–(d) Error types within each class; counts are submissions with at least one failing case of the type, and a submission can count under several types.

Prompt synthesis. InfraDev cannot distinguish models without prompt synthesis. We replace the synthesized prompt with the PR's original description on the Megatron-LM tasks (Figure 2). Once the hidden probes are included, neither GPT-6 Sol nor DeepSeek V4 Pro resolves a single task. Maintaining comprehensive AI infrastructure is hard, and a prompt that omits the required interfaces makes even capable agents fail for reasons unrelated to the task. With the synthesized prompt, the resolved rate is at most 52.6%; the prompt therefore does not hand the agent the implementation.

Resource-efficient Execution Environment. We compare our execution environment with building one Docker image per task. It reduces storage by 46% on Megatron-LM, from 37.3 to 20 GB, and by 42% on vLLM, from 115.9 to 67.5 GB. It also shortens environment build time by about 30% on Megatron-LM and 50% on vLLM.

4.4 Failure Analysis

Corpus and taxonomy. We analyze the 1,754 failing tests of the graded submissions of the five models other than GPT-6 Sol and group most of them into three error classes, from the most basic to the most demanding (Figure 3). The most basic is a **structural error**, in which the code does not fit the program around it because it fails to execute, breaks an API, mismatches a tensor's shape, layout, or dtype, or mishandles state or lifecycle. Once the code fits, a **numerical error** means it computes wrong values, because of a wrong algorithm or lost precision. Beyond correct values, a **behavioral error** means the code misses a requirement of the task, such as lowering peak memory or working under CUDA graphs. A submission counts under every class that appears among its failing tests.

Structural errors dominate, and state errors often escape official tests. Structural errors are the most common class, found in 48% of the submissions (Figure 3a). Within the class, execution and API errors lead, followed by state and lifecycle errors (Figure 3b). State errors often escape the official tests, which check the result of a single call rather than the state that the call leaves behind. In 15 of the 35 submissions with a state error, only our generated hidden probes catch it. Together, these errors show that models struggle most with changes that span many files of a large repository, where the code must fit the interfaces of many modules and manage state over a long lifetime.

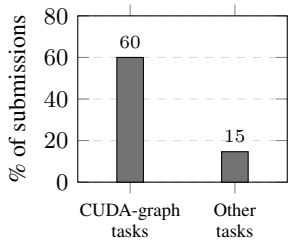


Figure 4: Share of submissions with a condition-specific error in different tasks.

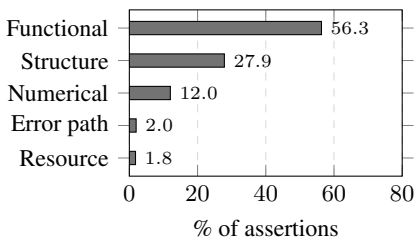


Figure 5: What the 5,532 assertions in agents' own tests check, over 94 traced runs.

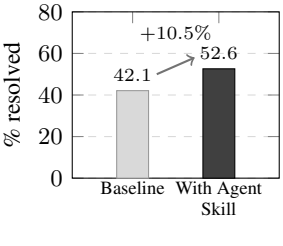


Figure 6: Resolved rate (%) of GPT-5.6 Sol on the 19 Megatron-LM tasks, with and without the agent skill.

Numerical errors come from wrong math, not lost precision. All 24 submissions with a numerical error contain an algorithm error, and none fails on precision alone (Figure 3c). The errors are far too large to be rounding noise, so they must come from wrong math rather than from precision lost to differences in hardware or implementation details. On Megatron-LM PR 2440, for example, one patch produces an indexer loss of 19.95 where the loss must be 0.0. Contrary to our own experience as infrastructure developers, where getting the algorithm right is relatively easy and matching numerical precision is the hard part, agents struggle with the algorithm rather than the precision.

Behavior breaks under execution modes such as CUDA graphs. Of the 51 submissions with a behavioral error, 35 contain an error that appears only under a specific execution mode (Figure 3d). As Figure 4 shows, 60% of the submissions on the four tasks built around CUDA graphs have such an error, compared with 15% on the other tasks. On the CUDA-graph tasks, every such error stays hidden in the default eager mode and appears only once CUDA graphs are turned on. Agents appear to write code for eager mode and treat CUDA graphs as a simple on/off switch, overlooking that a CUDA graph replays recorded GPU work without rerunning the surrounding Python code, so any logic that must run or change on every call is silently skipped.

Agents rarely test resource requirements like memory, time, or traffic. Agents write their own tests in 94 of the 103 traced runs, but fewer than 2% of the 5,532 assertions check resource usage (Figure 5). Most assertions check functional results (56%) or structure (28%). Of the 11 traced runs that fail a resource check, only one writes any resource assertion. Agents check that their code runs and returns the right output, but rarely check whether it delivers the efficiency requirement that the task was written for.

4.5 Improving Agents with Reusable Skills

Based on our analysis in Section 4.4, we build an agent skill for AI infrastructure development around three principles.

- **Check every caller and every piece of state the change touches.** Structural errors are the most common class. The skill asks the agent to find each caller of a function or class it modifies, keep the signatures, tensor shapes, and dtypes those callers expect, and test stateful objects such as caches and counters over several calls rather than one.
- **Compare every new computation with a naive reference.** Numerical errors all come from wrong algorithms. The skill asks the agent to write the formula directly in a few lines of plain PyTorch, compare its code with this reference on small inputs, and check invariants that follow from the math, such as a loss that must be zero or gradients that must be averaged rather than summed across data-parallel ranks.
- **Test resource requirements and CUDA-graph behavior.** Behavioral errors come from resource requirements and execution modes that agents rarely test. The skill asks the agent to measure peak memory, runtime, or bytes sent before and after its change whenever the task sets a resource requirement, and to rerun its tests with CUDA graphs enabled and check that the outputs match eager mode.

The skill contains no task-specific answers or hidden tests. On Megatron-LM tasks, it raises the resolved rate of GPT-5.6 Sol from 42.1% to 52.6%, a gain of 10.5 % points (Figure 6).

5 Conclusion and Future Work

We asked whether coding agents can develop real-world AI infrastructure, and how we can tell. The answer to the first is not yet: the best frontier agent resolves fewer than two in five tasks, and most of its failures are shared by every other agent. The answer to the second is that a pull request's own tests and description are not a reliable judge. The tests accept patches that skip what the pull request set out to do, and the description leaves out interfaces a correct patch depends on. Grading becomes reliable only when hidden tests and the prompt are written together. The failures are informative. Agents rarely get the algorithm wrong; they struggle to fit new code into a large existing program and to check that a change saves the memory, time, or traffic it promises. A skill that targets these weaknesses already helps. Because the synthesis pipeline is automatic, InfraDev can grow with new pull requests as the infrastructure itself evolves.

The main limitation of InfraDev is its size: selecting tasks requires careful manual review by domain experts, which limits the benchmark to 36 tasks. Because test and prompt synthesis is automatic, we will expand InfraDev to more tasks and repositories and update it continuously with newly merged pull requests and develop automatic methods for task selection.

AI use statement

In this work, we used generative AI tools, namely LLM coding agents, for the following tasks that require disclosure. They generated synthetic data: agent skills direct coding agents to synthesize the hidden probes, the adversarial patches, and the task prompts of Section 3.1.2. They implemented methods, including parts of the benchmark construction, the evaluation harness, and the analysis scripts. They cleaned and reformatted data: coding agents wrote the scripts that collect every graded submission with its failing tests, parse the agent traces, and convert both into the data behind the failure analysis. They helped develop the conceptual framework of the failure analysis by proposing candidate error types and helping the authors organize them into the structural, numerical, and behavioral classes of Section 4.4. They supported qualitative analysis, refined hypotheses, and interpreted results: coding agents clustered failing test cases, proposed labels under the rules of Appendix C, and checked candidate explanations of each finding, such as why some errors appear only under CUDA graphs, against the failure logs. They provided feedback on the research methodology by suggesting how to organize the agent skill of Section 4.5 around the three error classes and by helping draft its principles. They also assisted with translation, turning drafts that the authors wrote in Chinese into English. We have not used generative AI tools to select the pull requests, which we chose manually, and formulating mathematical claims or writing proofs is not applicable to this work. Additionally, we used generative AI tools to create the tables and figures and to draft and edit parts of the paper for readability. The models evaluated in this paper are the subjects of our study, not tools used to produce it. We have reviewed all AI-assisted work. Every hidden probe enters the benchmark only after it passes the executable acceptance check of Algorithm 2, and every task prompt only after a linter confirms that it names every symbol of the interface contract and a checker agent finds no disclosure of the solution. The authors reviewed the synthesized prompts, the failure labels, and all AI-edited text, and checked every reported number against the recorded run outputs. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation, 2025. URL https://arxiv.org/abs/2503.11926.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 83418–83452, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/86ba836d4c5dd859d795a172911745e2-Paper-Conference.pdf.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 50466–50494, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/7e3767db483c942b883eb4f8cfb74e31-Paper-Conference.pdf.

Willy Chan, Nathan Paek, Simon Guo, Simran Arora, and Daniel Y. Fu. ParallelKernelBench: Can LLMs write fast Multi-GPU kernels?, 2026. URL https://nathanjpaek.github.io/parallel-kernel-bench/.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI blog, https://openai.com/index/introducing-swe-bench-verified/, 2024.

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?, 2025. URL https://arxiv.org/abs/2509.16941.

Leilei Ding, Shumin Wang, Yuting Huang, Fanqi Wan, Yinmin Zhang, Qi Han, Yiming Xu, Feiyuan Zhang, Xiaomeng Chu, Guoliang You, Wuyang Zhang, Daxin Jiang, and Yanyong Zhang. Φ-Bench: Can large language models engineer the infrastructure that powers them?, 2026. URL https://arxiv.org/abs/2609.10226.

Yuan Gao, Zeren Yang, Junnan Li, Shawn (Wanxiang) Zhong, Ahmed Dajani, Mai Zheng, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. InfraBench: Evaluating infrastructure agents across layers, lifecycle, and risk. In *Workshop on Hot Topics in System Infrastructure (HotInfra)*, 2026. URL https://arxiv.org/abs/2608.11234. Co-located with ISCA '26.

Jinjun Huang, Zhongzhen Wen, Tongtong Xu, Meng Yan, Xin Xia, and Zhongxin Liu. RealisticTritonBench: A benchmark for Triton-kernel generation in real-world AI frameworks, 2026. URL https://arxiv.org/abs/2608.12004.

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 B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Miles, Seraphina Nix, Tao Lin, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring AI ability to complete long software tasks. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi,

and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 92213–92266. Curran Associates, Inc., 2025. doi: 10.52202/085713-3086. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/85069585133c4c168c865e65d72e9775-Paper-Conference.pdf.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph 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*, SOSP '23, pp. 611–626. ACM, October 2023. doi: 10.1145/3600006.3613165. URL http://dx.doi.org/10.1145/3600006.3613165.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization, 2025. URL https://arxiv.org/abs/2509.14279.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. TritonBench: Benchmarking large language model capabilities for generating Triton operators. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1183. URL https://aclanthology.org/2025.findings-acl.1183/.

Kevin Luu. Keeping vLLM production quality: A look inside CI, benchmarking, and the release process. vLLM Blog, July 2026. URL https://vllm.ai/blog/2026-07-16-keeping-vllm-production-quality.

Ayush Nangia, Shikhar Mishra, Aman Gokrani, and Paras Chopra. ISO-Bench: Can coding agents optimize real-world inference workloads?, 2026. URL https://arxiv.org/abs/2602.19594.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 1279–1297. ACM, March 2025. doi: 10.1145/3689031.3696075. URL http://dx.doi.org/10.1145/3689031.3696075.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. PaperBench: Evaluating AI's ability to replicate AI research. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 56843–56873. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/starace25a.html.

Natalia Tarasova, Enrique Balp-Straffon, Aleksei Iancheruk, Yevhenii Sielskyi, Nikita Kozodoi, Liam H. Byrne, Jack Butler, Dayuan Jiang, Marcin Czelej, Andrew Ang, Yash Shah, Roi Blanco, and Sergei Ivanov. SWE-InfraBench: Evaluating language models on cloud infrastructure code, 2026. URL https://arxiv.org/abs/2606.05249.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing Triton kernel AI agent & evaluation benchmarks, 2025a. URL https://arxiv.org/abs/2507.23194.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel 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 Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Hjalmar Wijk, Tao Roa Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Joshua M Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Jun Koba Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. RE-Bench: Evaluating frontier AI R&D capabilities of language model agents against human experts. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 66772–66832. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/wijk25a.html.

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 A. Chowdhery and Z. Jia (eds.), *Proceedings of Machine Learning and Systems*, volume 8, pp. 2016–2064. MLSys, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/file/37e44c4b5321605735be9761f9b758fc-Paper-Conference.pdf.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

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

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-Bench. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.189. URL https://aclanthology.org/2025.acl-long.189/.

Boxi Yu, Yang Cao, Yuzhong Zhang, Liting Lin, Junjielong Xu, Zhiqing Zhong, Qinghua Xu, Guancheng Wang, Jialun Cao, Shing-Chi Cheung, Pinjia He, and Lionel Briand. SWE-ABS: Adversarial benchmark strengthening exposes inflated success rates on test-based benchmark, 2026. URL https://arxiv.org/abs/2603.00520.

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 D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-2111. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/

5afa9cb1e917b898ad418216dc726fbd-Paper-Datasets_and_Benchmarks_Track.pdf.

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 A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

A Details of Test and Prompt Synthesis

A.1 Hidden-Probe Synthesis

Probe types. A *numerical probe* compares an output such as a loss, a gradient, or a model state with a reference taken from an existing implementation (Megatron-LM PR 3922 compares the cached Mamba state with a pre-PR path), an invariance (in PR 1989, changing the chunk size must not change the output), a value derived by hand (PR 2440 recomputes the sparse attention in float64 with plain loops), or a property that must always hold (a KL divergence of a distribution from itself is zero). A *behavioral probe* measures whether the promised effect occurs and compares it with a value derived from the task, such as the bytes that activation offloading moves (PR 1913) or the elements each rank receives (PR 3213).

Acceptance check. Algorithm 2 gives the check. Besides the pre-PR snapshot and the adversarial patches, every probe must reject a *hollow patch*, which keeps the interfaces of the golden patch but makes every function added by the PR raise an exception, so that no probe only checks that the new functions exist. The promise that each adversarial patch breaks is recorded by the agent that writes it and is not checked by execution. A PR whose probes do not pass within a fixed number of rounds is left without probes.

Algorithm 2 CheckProbes, the acceptance check of Algorithm 1. Every run uses the GPU count used for scoring, and a pass on Δ_{gold} requires that no case is skipped.

Require: hidden probes T_{probe} and adversarial patches $\Delta_1, \ldots, \Delta_k$; from Algorithm 1, the PR r, its golden patch Δ_{gold}, the pre-PR snapshot $\mathcal{S}$, and the official tests T_{off}
Ensure: *true* if and only if T_{probe} and $\Delta_1, \ldots, \Delta_k$ are accepted

function CheckProbes($T_{\text{probe}}, \Delta_1, \ldots, \Delta_k$)
  $\Delta_{\text{hollow}} \leftarrow \Delta_{\text{gold}}$ with every function added by r raising an exception
  $ok_1 \leftarrow$ every probe passes on Δ_{gold} in three consecutive runs ▷ accepts the right patch
  $ok_2 \leftarrow$ every probe fails on $\mathcal{S}$, Δ_{hollow}, and $\Delta_1, \ldots, \Delta_k$ ▷ rejects wrong ones
  $ok_3 \leftarrow \Delta_1, \ldots, \Delta_k$ score no worse than Δ_{gold} on T_{off} ▷ they fool T_{off}
  return $ok_1 \wedge ok_2 \wedge ok_3$

B Detailed Performance Tables

Table 4 reports the results of Table 2 for each repository separately, together with the mean number of model turns. Table 5 gives the resolved rate of each model under the official tests alone and with the hidden probes, per repository. Table 6 splits each token total into non-cached input, cache reads, output, and reasoning.

Table 4: Table 2 per repository. Steps is the mean number of completed model turns per task.

Model	% Resolved	% Passed	Time (min)	Steps	Cost ($)	Tokens (M)
Megatron-LM: 19 tasks						
GPT-6 Sol	**52.6**	80.6	**50.4**	227.6	6.66	**27.00**
GPT-5.6 Sol	**42.1**	80.1	**71.6**	**163.4**	16.90	**22.67**
GLM 5.3	26.3	**84.4**	149.5	203.2	12.79	45.44
HY4 preview	**42.1**	**91.0**	137.8	199.5	**2.14**	32.35
DeepSeek V4 Pro	**42.1**	78.0	103.2	186.6	**2.23**	42.66
Kimi K3	31.6	77.0	148.1	**157.4**	13.46	31.79
verl: 7 tasks						
GPT-6 Sol	**42.9**	**91.0**	**30.7**	**88.1**	3.72	**12.31**
GPT-5.6 Sol	28.6	83.8	52.3	**114.6**	11.23	**15.05**
GLM 5.3	**42.9**	78.4	101.8	155.8	9.32	32.69
HY4 preview	**57.1**	88.9	53.7	162.0	**1.43**	22.16
DeepSeek V4 Pro	28.6	78.5	**42.6**	138.6	**1.26**	26.04
Kimi K3	**57.1**	**92.1**	93.6	160.0	10.47	28.12
vLLM: 10 tasks						
GPT-6 Sol	10.0	**66.8**	**40.8**	**150.6**	4.07	**15.18**
GPT-5.6 Sol	10.0	**72.2**	**93.3**	246.7	25.38	**36.31**
GLM 5.3	10.0	61.1	134.2	260.6	18.71	64.99
HY4 preview	10.0	64.6	125.8	214.9	**2.47**	37.32
DeepSeek V4 Pro	10.0	64.3	112.7	215.0	**1.69**	51.21
Kimi K3	10.0	62.2	144.0	**176.1**	15.62	40.80

Table 5: Resolved rate (%) under the official tests alone and under the official tests plus the supplementary probes. A task is resolved when every test in the suite passes.

Model	Official tests	Official + probes
Megatron-LM: 19 tasks		
GPT-6 Sol	68.4	52.6
GPT-5.6 Sol	52.6	42.1
GLM 5.3	36.8	26.3
HY4 preview	52.6	42.1
DeepSeek V4 Pro	52.6	42.1
Kimi K3	42.1	31.6
verl: 7 tasks		
GPT-6 Sol	100.0	42.9
GPT-5.6 Sol	85.7	28.6
GLM 5.3	71.4	42.9
HY4 preview	85.7	57.1
DeepSeek V4 Pro	71.4	28.6
Kimi K3	85.7	57.1
vLLM: 10 tasks		
GPT-6 Sol	10.0	10.0
GPT-5.6 Sol	20.0	10.0
GLM 5.3	20.0	10.0
HY4 preview	10.0	10.0
DeepSeek V4 Pro	10.0	10.0
Kimi K3	20.0	10.0

Table 6: Mean parent-session tokens in millions. Input excludes cached tokens. The four components sum to the token totals in Table 4 before rounding and average the same runs.

Model	Input	Cache read	Output	Reasoning
Megatron-LM				
GPT-6 Sol	0.336	26.600	0.037	0.030
GPT-5.6 Sol	0.768	21.832	0.042	0.029
GLM 5.3	0.485	44.813	0.055	0.086
HY4 preview	0.622	31.608	0.046	0.072
DeepSeek V4 Pro	1.580	40.930	0.061	0.087
Kimi K3	1.037	30.645	0.050	0.056
verl				
GPT-6 Sol	0.362	11.887	0.038	0.024
GPT-5.6 Sol	0.441	14.555	0.035	0.023
GLM 5.3	0.280	32.285	0.046	0.074
HY4 preview	0.333	21.736	0.045	0.050
DeepSeek V4 Pro	1.031	24.889	0.047	0.069
Kimi K3	0.279	27.745	0.055	0.045
vLLM				
GPT-6 Sol	0.255	14.870	0.033	0.026
GPT-5.6 Sol	0.948	35.262	0.056	0.044
GLM 5.3	1.249	63.574	0.066	0.099
HY4 preview	0.772	36.428	0.036	0.083
DeepSeek V4 Pro	0.863	50.189	0.067	0.089
Kimi K3	0.671	40.000	0.054	0.077

C Failure Analysis: Error Types and Labelling

Failing cases and filtering. The analysis covers the $36 \times 5 = 180$ submissions of the five models other than GPT-6 Sol. A failing case is a test that the reference passes and the submission does not, and the remaining submissions contain 1,754 such cases. We group them by task, test file, and error signature, and label each group by reading the failing assertion against the prompt and the submission's code. We first remove failures that are not the agent's fault: *specification gaps*, where a test depends on a name or convention that the prompt neither states nor implies; *fixture failures*, which depend on a test double that the prompt does not constrain; and *environment failures*, which the reference also shows on the same hardware. Each remaining case receives one label: an exception is structural, and otherwise the property that the failing assertion checks decides the label. Table 7 counts the submissions with at least one case of each type.

Structural errors. The code does not fit the program around it. An *execution or API* error stops the code before any value can be checked: an import or collection failure, a missing symbol, a signature that does not match its callers, or a crash. A *shape, layout, or dtype* error produces a tensor whose shape, memory layout, or data type differs from what the surrounding code expects. A *state or lifecycle* error mishandles data that persists across calls, such as caches, buffers, counters, and the ownership of memory, or mishandles the capture and replay of CUDA graphs.

Behavioral errors. The code computes correct values but misses a requirement of the task. A *resource* error misses a stated requirement on memory, speed, communication traffic, or liveness, such as a peak memory that does not drop or a weight sync that sends more entries than have changed. A *condition-specific* error appears only under a particular condition: a flag, a parallel or CUDA-graph mode, a boundary input, a fallback, or an error path. A numerical mismatch that appears only under such a condition is labelled condition-specific rather than numerical.

Numerical errors. The code runs and fits the program but computes wrong values. An *algorithm* error implements the wrong computation, such as summing gradients across data-parallel ranks instead of averaging them. A *precision-only* error implements the right computation with insufficient precision; we assign it only when the same patch passes an exact-logic check.

Table 7: Submissions exhibiting each error type, counted once per submission. A submission may exhibit several types. An exception or collection failure is structural; otherwise a failure is labelled by the property its assertion checks.

Class	Error type	Megatron-LM	verl	vLLM	All
Structural	Execution / API	19	10	19	48
Structural	Shape / layout / dtype	8	4	10	22
Structural	State / lifecycle	10	8	17	35
Structural	**Any structural error**	**30**	**18**	**37**	**85**
Behavioral	Resource (memory, speed, traffic)	14	1	2	17
Behavioral	Condition-specific behavior	9	0	26	35
Behavioral	**Any behavioral error**	**22**	**1**	**28**	**51**
Numerical	Algorithm	7	6	11	24
Numerical	Precision only	0	0	0	0
Numerical	**Any numerical error**	**7**	**6**	**11**	**24**