SWE-GPU-Bench: Can Language Models Solve Real-World GPU Software Engineering Tasks?

Feng Chen **Binbin Liu** **Wenhan Han** **Yin Zheng***
Independent Researcher
yzheng3xg@gmail.com

Abstract

Existing benchmarks for GPU programming with language models evaluate whether a model can generate an isolated kernel, operator, or standalone CUDA program. Most GPU software, however, is developed inside existing repositories, where a single change often spans CUDA kernels, C++ host code, Python bindings, build files, and tests. We introduce SWE-GPU-Bench, a benchmark of 417 tasks drawn from merged pull requests in 30 GPU repositories, including CUTLASS, CCCL, FlashInfer, Triton, and TensorFlow. Working in a pre-configured environment, a model receives a problem statement and the repository at the base commit to generate a patch, which we evaluate using the pull request's test commands. The tasks cover bug fixes, feature implementations, and performance optimizations, with a small subset of performance tasks (Perf-Replay) also including benchmark commands that measure the speedup of patches that pass the correctness tests. Over a quarter of the tasks edit more than four files, and over half edit files in two or more languages. We evaluate frontier open-weight and proprietary models in both the oracle and agent settings. The strongest evaluated configurations, GPT-6-Astra and GPT-6.1-Sol in Codex CLI, resolve 58.3% and 58.0% of tasks, respectively.

1 Introduction

Recent benchmarks evaluate language models on GPU code generation. KernelBench and TritonBench, for example, test whether a model can generate a functionally correct and performant kernel or operator from a specification [Ouyang et al., 2025a, Li et al., 2025a]. In practice, however, GPU software engineering takes place inside large repositories, where updating a kernel often requires coordinated changes across host launch logic, language bindings, build configurations, and tests.

Repository-level benchmarks such as SWE-bench capture software maintenance for Python projects [Jimenez et al., 2024], but GPU repositories introduce two further challenges. First, their codebases span multiple languages and build systems, with CUDA and C++ sources compiled through CMake, Bazel, or setuptools and invoked from Python. Second, executing tests requires matching the target CUDA toolkit, architecture-specific compiler flags, and GPU hardware. Addressing these cross-language build dependencies and hardware execution requirements is central to real-world GPU engineering, yet existing repository-level GPU benchmarks remain limited to small-scale code translation [Davis et al., 2025] or narrow inference-engine tuning [Nangia et al., 2026], rather than general software maintenance across bug fixes, feature implementations, and performance optimizations.

We introduce SWE-GPU-Bench (Figure 1), constructing tasks from merged pull requests (PRs) across 30 GPU repositories.[2] For each task, we first run task-specific setup commands to prepare the build

*Corresponding author.

[2]Data: https://huggingface.co/datasets/Froze/SWE-GPU-Bench. Code: https://github.com/chenfengshijie/SWE-GPU-Bench. We report the current release of 417 tasks (378 general-set and 39 Perf-Replay tasks); the benchmark dataset is actively maintained and will be continuously updated.

40th Conference on Neural Information Processing Systems (NeurIPS 2026). Track on Evaluations and Datasets.

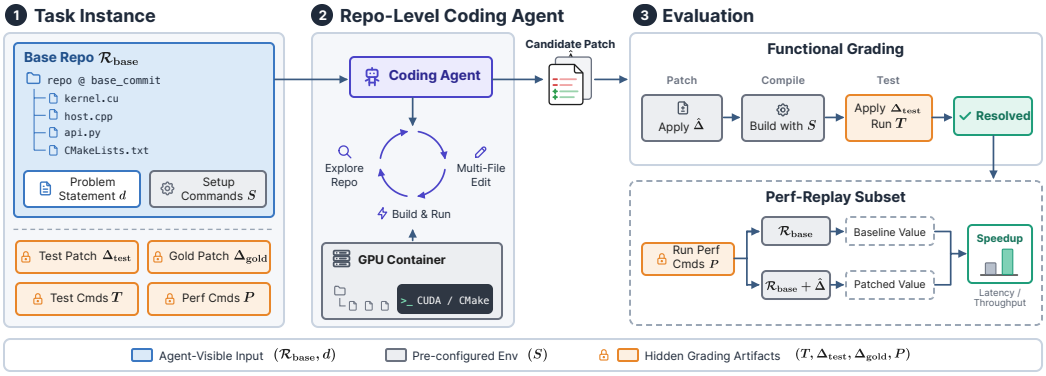


Figure 1: Overview of SWE-GPU-Bench evaluation. ① Each task instance is a tuple $(\mathcal{R}_{\text{base}}, d, S, T, P, \Delta_{\text{test}}, \Delta_{\text{gold}})$. After the environment is prepared with S, the model or agent receives only the repository $\mathcal{R}_{\text{base}}$ at the base commit and the problem statement d, while T, Δ_{test}, Δ_{gold}, and P remain hidden. ② A coding agent explores the repository, edits files, and builds and runs code in a GPU container to output a candidate patch $\hat{\Delta}$. ③ Functional grading applies $\hat{\Delta}$ and Δ_{test} and runs T; for Perf-Replay tasks, patches passing T are benchmarked with P against $\mathcal{R}_{\text{base}}$ to measure speedup.

environment, then provide the model with the repository at the base commit and a problem statement distilled from the PR, and finally apply the PR's test changes and run its test commands to grade the generated patch. To construct the benchmark, we filter candidate PRs with deterministic rules and an LLM judge, use a coding agent to synthesize setup and test commands, and retain only tasks whose tests fail before applying the gold patch and pass afterward, yielding 417 tasks. For a subset of performance tasks (Perf-Replay), we also synthesize benchmark commands to measure the speedup of patches that pass functional tests.

We evaluate frontier open-weight and proprietary models under two settings following SWE-bench and SWE-Perf [Jimenez et al., 2024, He et al., 2026]: an *oracle* setting, where a model receives the files modified by the gold patch and generates a patch in a single pass, and an *agent* setting, where a coding agent iteratively explores the repository, edits code, and invokes the build and test toolchain. Across evaluated models, iterative agent interaction yields substantially higher resolution rates than single-pass oracle generation even when the oracle is given the exact target files. Whereas earlier models frequently fail to compile across heterogeneous GPU build systems, the latest frontier models achieve nearly 100% compilation accuracy—reflecting rapid progress on repository-level GPU builds—yet the strongest configurations, GPT-6-Astra and GPT-6.1-Sol in Codex CLI, resolve 58.3% and 58.0% of tasks, with remaining failures concentrated in test execution.

Our contributions are:

- **A realistic repository-level GPU benchmark.** We introduce SWE-GPU-Bench, spanning bug fixes, feature implementations, and performance optimizations across production GPU libraries, compilers, and frameworks. Its tasks require coordinated edits across CUDA kernels, C++ host logic, Python bindings, and heterogeneous build systems, complemented by the Perf-Replay subset for correctness-gated speedup evaluation.
- **An automated, execution-verified construction pipeline.** To handle heterogeneous GPU build environments, our five-stage pipeline combines rule- and LLM-based filtering with an interactive coding agent that synthesizes task-specific setup, test, and benchmark commands inside GPU containers. Every task is verified via strict fail-to-pass replay in a clean environment and audited by human reviewers against solution leakage.
- **A multi-stage empirical evaluation and failure analysis.** Evaluating frontier open-weight and proprietary models across oracle and agent settings, we decompose performance into patch application, compilation, test execution, and runtime speedup. Our analysis demonstrates the necessity of execution feedback over static file localization, quantifies accuracy degradation across multi-file and multi-language edits, and reveals a sharp gap between functional correctness and end-to-end GPU optimization.

2 SWE-GPU-Bench

SWE-GPU-Bench is a repository-level benchmark of 417 GPU software maintenance tasks constructed from merged PRs, encompassing a general set of 378 tasks for functional evaluation and a Perf-Replay subset of 39 tasks for runtime speedup evaluation (Section 2.4).

2.1 Task Formulation

Each task is packaged as $(\mathcal{R}_{\text{base}}, d, S, T, P, \Delta_{\text{test}}, \Delta_{\text{gold}})$. We first check out the repository $\mathcal{R}_{\text{base}}$ at the base commit of the PR, remove all subsequent git history, and run the setup commands S to prepare the build environment. Providing a pre-configured environment decouples repository-level code maintenance from container provisioning. The model or agent is then given $\mathcal{R}_{\text{base}}$ and the problem statement d and outputs a candidate patch $\hat{\Delta}$. The grading artifacts remain hidden from the model: the test patch Δ_{test} (tests added or modified by the PR), the gold patch Δ_{gold} (all remaining changes), the test commands T, and, for Perf-Replay tasks, the benchmark commands P.

To evaluate $\hat{\Delta}$, we apply $\hat{\Delta}$ to $\mathcal{R}_{\text{base}}$, revert any candidate modifications to files targeted by Δ_{test}, apply Δ_{test}, and run T to compile and execute the PR's tests. A task is marked *resolved* if all commands in T exit with code 0, consistent with the fail-to-pass verification during construction (Section 2.2). For Perf-Replay tasks that pass T, we additionally run P to evaluate runtime speedup over $\mathcal{R}_{\text{base}}$ (Section 2.4). Over all attempted tasks, we report three metrics: **Patch** (producing a non-empty, cleanly applicable patch), **Compile** (building the patched repository without compilation errors), and **Test** (passing T to resolve the task).

2.2 Benchmark Construction

Because many PRs in GPU repositories lack targeted tests, bundle unrelated changes, or depend on custom CI environments, we construct tasks through a five-stage pipeline (Figure 2); Appendix A provides full filtering criteria and benchmark statistics.

Stage I: Repository selection and PR collection. We select actively maintained GPU repositories from official organizations and core maintainers, spanning CUDA libraries, GPU compilers such as Triton and TVM, ML frameworks, and inference engines. Across these repositories, we collect merged PRs along with their metadata, commit histories, and diffs.

Stage II: Filtering. Deterministic rule filters retain PRs that merge into the default branch, reference a resolved issue, contain no binary diffs, and modify both tests and at least one CUDA, C++, or GPU build file. To focus on self-contained maintenance tasks with isolatable fail-to-pass tests, we also bound the number of modified files and lines, excluding broad refactors and release merges (Appendix A.1). An LLM judge (GPT-5.2) then classifies each remaining PR (bug fix, feature, or performance) and scores its GPU relevance, problem clarity, test linkage, and hardware reproducibility on NVIDIA A100 or H100 GPUs. Candidates rated accepted or borderline advance to synthesis; the LLM judge serves strictly as a candidate filter, whereas final task admission requires executable verification (Stage IV) and manual audit (Stage V). Each candidate's diff is partitioned into the test patch Δ_{test} (test files and test-support targets) and the gold patch Δ_{gold} (all remaining changes).

Stage III: Command synthesis. For each candidate PR, a synthesis agent runs inside an A100 or H100 GPU container matched to the target architecture (e.g., H100 for Hopper-specific features such as `sm_90a`). The agent alternates between inspecting repository metadata, build scripts, and CI workflows and executing trial builds and test runs, iteratively refining its commands until it synthesizes setup commands S, test commands T targeting the PR's tests, and, for performance PRs, benchmark commands P. Command synthesis is the most compute-intensive stage of construction (prompt in Appendix E.3; cost details in Appendix B.2).

Stage IV: Clean replay. We replay S and T in a clean, agent-free GPU container. A candidate is admitted only if T fails on $\mathcal{R}_{\text{base}}$ with Δ_{test} applied and passes with exit code 0 once Δ_{gold} is also applied. For performance PRs, we additionally execute P on $\mathcal{R}_{\text{base}}$ and $\mathcal{R}_{\text{base}} + \Delta_{\text{gold}}$ to confirm that the reference patch yields a measurable improvement (Section 2.4).

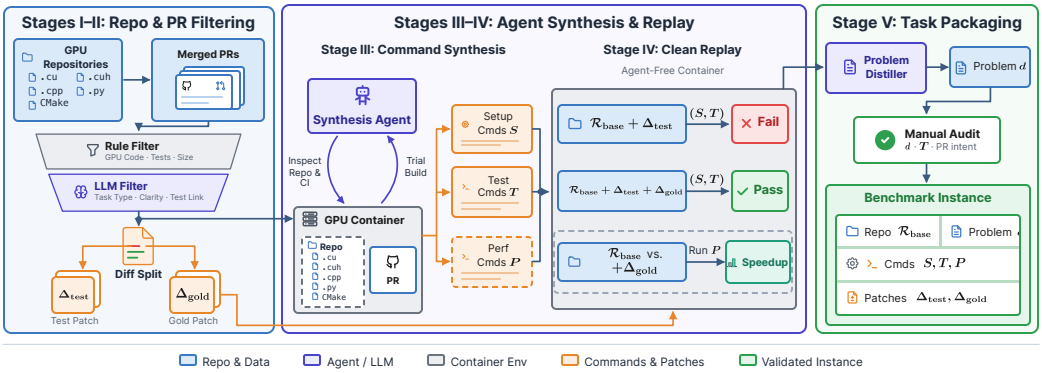


Figure 2: Construction pipeline of SWE-GPU-Bench. **Stages I–II (repository and PR filtering):** merged PRs from GPU repositories pass a rule filter (default-branch merge, linked issue, GPU code, tests, bounded code size) and an LLM filter (task type, clarity, test linkage), and each PR diff is split into the test patch Δ_{test} and the gold patch Δ_{gold}. **Stage III (command synthesis):** a synthesis agent in a GPU container alternates between inspecting the repository and CI and running trial builds to produce setup commands S, test commands T, and, for performance PRs, benchmark commands P. **Stage IV (clean replay):** in an agent-free container, (S, T) must fail on $\mathcal{R}_{\text{base}}+\Delta_{\text{test}}$ and pass once Δ_{gold} is also applied; for performance PRs, P is additionally run on $\mathcal{R}_{\text{base}}$ and $\mathcal{R}_{\text{base}}+\Delta_{\text{gold}}$ to verify the reference speedup. **Stage V (task packaging):** an LLM distills the problem statement d, a manual audit verifies d, T, and the PR intent, and the result is packaged as $(\mathcal{R}_{\text{base}}, d, S, T, P, \Delta_{\text{test}}, \Delta_{\text{gold}})$. Dashed elements apply only to performance PRs.

Stage V: Task packaging. An LLM distills the problem statement d from the PR title, description, and diff under constraints that prohibit naming modified files, internal symbols, or edit steps unless required by public test interfaces (Appendix E.2). Human reviewers then audit every candidate passing clean replay to ensure that d leaks no solution details, that T exercises the intended change, and that d, T, and the PR intent are mutually consistent (Appendix A.4). Accepted candidates are packaged as $(\mathcal{R}_{\text{base}}, d, S, T, P, \Delta_{\text{test}}, \Delta_{\text{gold}})$ (format in Appendix A.2), with P included for Perf-Replay tasks.

2.3 Features of SWE-GPU-Bench

Production GPU repositories. Tasks are drawn from 30 repositories (Figure 3) spanning four layers of the GPU software stack: CUDA libraries (CCCL, CUTLASS, RAFT), compilers and DSLs (Triton, TVM, XLA, TileLang), ML frameworks (TensorFlow, JAX, DeepSpeed, Megatron-LM), and inference engines (FlashInfer, LMDeploy, llama.cpp). Models must therefore localize edits within large-scale, production-grade repositories.

Multi-file, cross-language edits. Gold patches frequently require coordinated changes across files and language boundaries: over a quarter of the tasks modify more than four files, and more than half span two or more languages—typically coupling low-level CUDA or C++ device code with host bindings in Python and build configurations in CMake or Bazel (Figure 4a).

Diverse maintenance categories. Following the primary classification from Stage II, the benchmark spans bug fixes, feature implementations, and performance optimizations (Figure 4b), with hardware-compatibility updates, build-infrastructure changes, and refactorings grouped under features.

Architecture-aware executable verification. Every task packages executable setup and test commands that compile the affected targets and run the PR's tests inside a GPU container (Table 1). Typical workflows configure CMake, Bazel, or custom build scripts for a specific compute capability (such as `sm_80` or `sm_90a`) and invoke targeted GoogleTest or pytest suites, as illustrated by the CUTLASS case study in Appendix A.5.

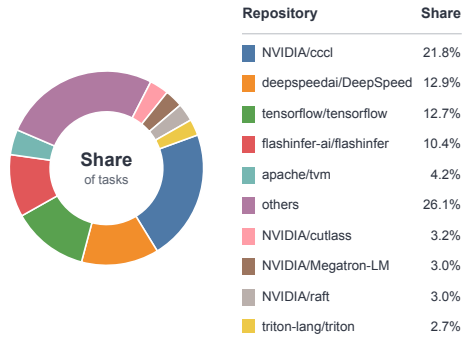
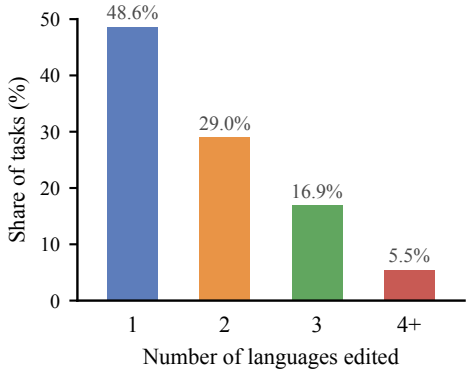


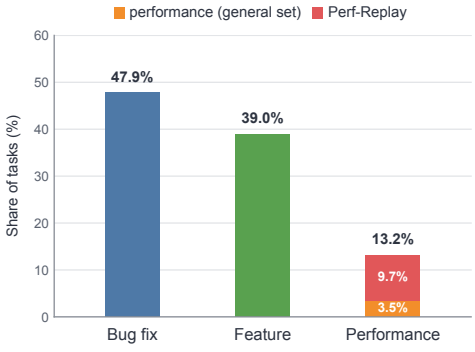
Figure 3: Distribution of tasks by repository, with the smallest shares grouped as *others*.

Table 1: Mean and maximum of task attributes over all tasks. Appendix A.3 reports percentiles.

		Mean	Max
Problem statement	# Words	236.6	2,636
Gold patch	# Files	3.7	62
	# Functions	6.0	166
	# Lines	178.6	3,227
Test patch	# Files	1.8	13
	# Lines	126.0	927
Test commands	# Lines	6.8	102



(a) Languages per gold patch.



(b) Task types.

Figure 4: Composition of SWE-GPU-Bench, in percent of tasks. (a) Number of languages edited by the gold patch. (b) Distribution of task types, with the performance bar split into general-set and Perf-Replay tasks.

2.4 Perf-Replay

For performance-optimization PRs, unit tests verify functional correctness rather than runtime speedup. For the subset of performance PRs that include a reproducible benchmark workload, we construct Perf-Replay by synthesizing benchmark commands P configured to output a scalar value for the target performance metric (such as latency, throughput, or kernel execution time). Performance tasks without a standalone benchmark remain in the general set and are evaluated through their correctness tests T.

Evaluating a candidate patch on Perf-Replay applies a correctness gate: the patch must first pass all functional tests in T. For patches that pass T, we execute P on both $\mathcal{R}_{\text{base}}$ and the patched repository to obtain their scalar measurements and compute the relative improvement ratio over $\mathcal{R}_{\text{base}}$. Because standardized performance evaluation across heterogeneous GPU workloads and metrics remains an open challenge, we report task-level relative changes to support correctness-gated performance analysis.

3 Experimental Setup

Oracle setting. Following the oracle retrieval setting of SWE-bench and the oracle setting of SWE-Perf [Jimenez et al., 2024, He et al., 2026], we provide the model with the problem statement, the repository README, and the full contents of the non-test files modified by the gold patch. The model outputs search-and-replace edits in a single pass, which are applied via exact block matching and converted into a unified diff (prompt in Appendix E.4). Providing the target files isolates edit

generation from repository-wide localization, offering a controlled reference setting in which the edit locations are known.

Agent setting. We run the GPT models in Codex CLI and all other models in OpenCode (prompt in Appendix E.5). Operating in the pre-configured container environment (Section 2.1), each agent is given a single run per task to autonomously inspect, edit, build, and test the repository, after which we record its final `git diff` as the candidate patch $\hat{\Delta}$.

Models and evaluation protocol. Table 2 lists the evaluated models, and Appendix B.1 provides their model identifiers, harness configurations, and evaluation coverage across dataset revisions. Models in both the oracle and agent settings are evaluated on a single NVIDIA A100 or H100 GPU matched to each task's target architecture under a single pass@1 run per configuration.

4 Results

4.1 Main Evaluation Results

Table 2 summarizes the main evaluation results across the three pipeline stages (**Patch**, **Compile**, and **Test**). Among agent configurations, GPT-6-Astra and GPT-6.1-Sol in Codex CLI achieve the highest resolution rates (**Test**) at 58.3% and 58.0%, followed by GPT-5.6-Sol in Codex CLI at 56.7% and DeepSeek-V4.1-Flash in OpenCode at 55.3%. In the oracle setting, GPT-6-Astra achieves the highest resolution rate at 49.2%, followed by GPT-6.1-Sol at 47.6% and GPT-5.6-Sol at 38.5%.

Table 2: Main results on SWE-GPU-Bench. **Patch**: producing a non-empty, applicable patch; **Compile**: building the patched repository without compilation errors; **Test**: passing the test commands T to resolve the task. Best value per setting in bold.

Setting	Model	Patch (%)	Compile (%)	Test (%)
Oracle	GPT-6-Astra	**100.0**	**100.0**	**49.2**
	GPT-6.1-Sol	**100.0**	**100.0**	47.6
	GPT-5.6-Sol	**100.0**	**100.0**	38.5
	GPT-5.2	79.2	48.3	26.0
	DeepSeek-V4.1-Flash	**100.0**	**100.0**	24.3
	DeepSeek-V4-Pro	**100.0**	**100.0**	21.4
	Claude Sonnet 4	75.2	49.6	20.8
	GPT-5-Codex	77.1	49.2	20.4
	GLM-5	60.3	36.8	17.2
	Kimi K2.5	68.4	42.1	14.4
	GPT-5.1-Codex	52.2	31.6	12.9
	DeepSeek-V3.1	51.7	36.4	11.0
	DeepSeek-V4-Flash	75.6	68.9	8.6
	DeepSeek-R1	54.1	38.8	8.6
	DeepSeek-V3.2	41.6	25.8	8.1
	Gemini 2.5 Pro	19.1	10.5	4.8
Agent (Codex)	GPT-6-Astra	**100.0**	**100.0**	**58.3**
	GPT-6.1-Sol	**100.0**	**100.0**	58.0
	GPT-5.6-Sol	98.0	98.0	56.7
	GPT-5.5	**100.0**	95.2	48.9
	GPT-5-Codex	85.0	47.0	33.0
	GPT-5.1-Codex	74.7	57.8	33.0
	GPT-5 mini	87.8	56.7	30.0
Agent (OpenCode)	DeepSeek-V4.1-Flash	**100.0**	**100.0**	55.3
	DeepSeek-V4-Pro	94.0	94.0	43.3
	DeepSeek-V3.1	85.1	58.5	29.0
	Kimi K2.5	85.0	53.0	27.0
	DeepSeek-V3.2	86.7	56.1	26.5
	DeepSeek-V4-Flash	88.2	88.2	26.3

Models achieve higher resolution rates in the agent setting. Every model evaluated in both settings achieves a substantially higher resolution rate when run as an interactive agent than under single-pass oracle generation, even though the oracle is given the exact files modified by the gold patch. Specifically, GPT-6-Astra increases from 49.2% to 58.3%, GPT-6.1-Sol from 47.6% to 58.0%, GPT-5.6-Sol from 38.5% to 56.7%, GPT-5.1-Codex from 12.9% to 33.0%, and DeepSeek-V3.2 from 8.1% to 26.5%. The gain from iterative execution feedback is especially pronounced for the DeepSeek-V4 models: in single-pass oracle generation without compiler or test feedback, DeepSeek-V4.1-Flash (24.3%) and DeepSeek-V4-Pro (21.4%) trail GPT-6-Astra (49.2%) by more than 24 percentage points, whereas in the agent setting DeepSeek-V4.1-Flash more than doubles its resolution rate to 55.3% (and DeepSeek-V4-Pro to 43.3%), narrowing the gap with the leading GPT agents to within 3 percentage points.

Recent frontier models reach nearly 100% compilation accuracy, shifting the bottleneck to test execution. Across earlier model generations, a large fraction of generated patches fail to compile (Table 2), with compilation rates reaching at most 49.6% among standard earlier models in the oracle setting (e.g., 48.3% for GPT-5.2 and 10.5% for Gemini 2.5 Pro) and 58.5% in the agent setting. By contrast, the latest models—GPT-6-Astra, GPT-6.1-Sol, GPT-5.6-Sol, DeepSeek-V4.1-Flash, and DeepSeek-V4-Pro, along with GPT-5.5—produce applicable patches on 94.0%–100.0% of tasks and reach nearly 100% compilation accuracy across both settings (100.0% for GPT-6-Astra, GPT-6.1-Sol, and DeepSeek-V4.1-Flash). This sharp improvement across model generations reflects rapid progress in handling CUDA/C++ syntax, cross-file type consistency, and heterogeneous GPU build systems, largely eliminating build failures and leaving functional test verification as the primary bottleneck.

4.2 Fine-Grained Breakdown

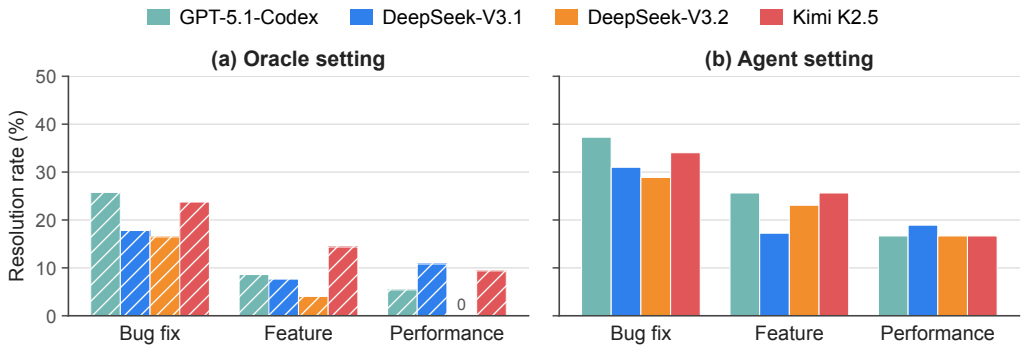


Figure 5: Resolution rate by task type in the oracle (a) and agent (b) settings, for four representative models evaluated in both settings. Appendix C lists the values.

Feature and performance tasks have lower resolution rates than bug fixes. In both settings, all four models shown in Figure 5 resolve a smaller fraction of feature and performance tasks than bug fixes. Kimi K2.5, for instance, resolves 23.8% of bug fixes versus 14.4% of feature tasks in the oracle setting, and 34.0% versus 25.6% in the agent setting. Whereas bug fixes often reduce to localized boundary checks or numerical corrections, feature implementations and performance optimizations typically require adding new template instantiations, dispatch paths, or host bindings across multiple components.

Resolution rates drop on patches that modify five or more files. In the oracle setting, resolution rates drop sharply on tasks that edit five or more files compared with tasks that edit one or two files, with DeepSeek-V3.1 dropping from 17.7% to 4.0% (Figure 6). Without compiler feedback, single-pass edits across many files frequently introduce cross-file signature or type mismatches. In the agent setting, resolution rates also decline on the 5+ file bin (for example, GPT-5.1-Codex drops from 37.5% on 1–2 files to 21.4% on 5+ files, compared with 24.6% to 4.8% in the oracle setting).

Single-language tasks achieve higher resolution rates. Across both settings, models generally achieve higher resolution rates on tasks that modify a single language than on multi-language

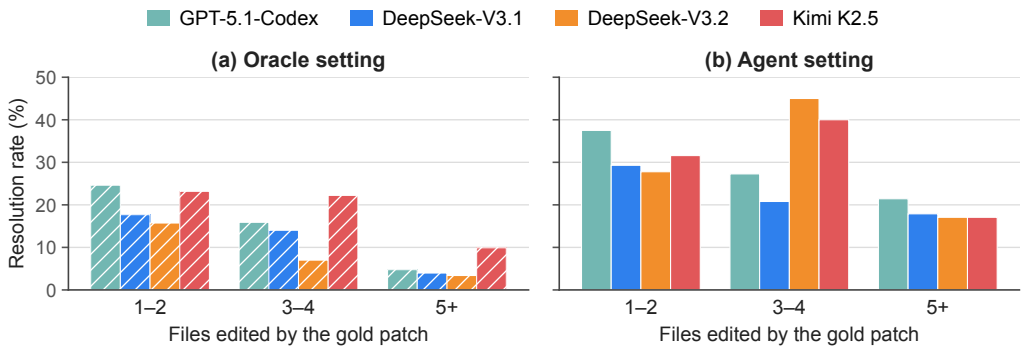


Figure 6: Resolution rate by the number of files that the gold patch edits, in the oracle (a) and agent (b) settings.

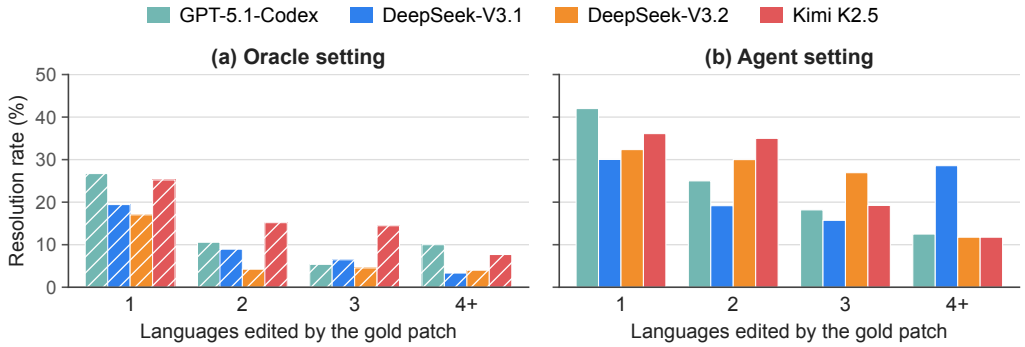


Figure 7: Resolution rate by the number of languages that the gold patch edits, in the oracle (a) and agent (b) settings.

tasks (Figure 7). In the oracle setting, resolution rates decline substantially from single-language tasks to tasks spanning four or more languages; in the agent setting, multi-language tasks—which require keeping device kernels, C++ host wrappers, Python interfaces, and build configurations synchronized—also score lower than single-language tasks.

4.3 Perf-Replay Evaluation

Table 3: Agent results on the Perf-Replay subset. **Patch**, **Compile**, and **Test** report task counts, and **Perf. gain** reports the mean relative improvement of the target scalar metric over the base commit across the tasks that pass **Test**.

Model	Patch	Compile	Test	Perf. gain (%)
Gold patch	39/39	39/39	39/39	6.4
GPT-5.1-Codex (Codex)	39/39	15/39	3/39	3.3
GPT-5-Codex (Codex)	39/39	10/39	2/39	2.4
Kimi K2.5 (OpenCode)	35/39	8/39	2/39	3.3
DeepSeek-V3.2 (OpenCode)	39/39	12/39	1/39	4.3

Passing the correctness gate remains the primary bottleneck on Perf-Replay. On the Perf-Replay subset, evaluated agents generate applicable patches for most tasks, but many fail during compilation and only a small fraction pass the functional correctness tests T (Table 3). Because each agent resolves only a few tasks—and not necessarily the same subset of tasks—the resulting **Perf. gain** values are computed over small, model-specific passing sets and should be interpreted as preliminary

observations rather than direct cross-model speedup rankings. On the tasks they do resolve, candidate patches achieve modest relative improvements over the base commit; Appendix C.1 examines representative case studies across models to illustrate how workload shapes, metric definitions, and patch implementations affect measured speedups.

5 Related Work

Table 4: Comparison of SWE-GPU-Bench with representative GPU kernel and repository-level software engineering benchmarks. **Repo-level**: tasks require editing an existing multi-file repository; **Real PRs**: tasks are derived from merged pull requests or commits; **Multi-lang. & build**: tasks span coordinated edits across device code, host bindings, and build configurations (e.g., CUDA/C++/Python/CMake); **GPU exec.**: patches are compiled and validated on GPU hardware.

Benchmark	Repo-level	Real PRs	Multi-lang. & build	GPU exec.	Task scope
KernelBench [Ouyang et al., 2025a]	–	–	–	✓	Isolated kernel generation
TritonBench [Li et al., 2025a]	–	–	–	✓	Isolated operator generation
FlashInfer-Bench [Xing et al., 2026]	–	–	–	✓	Kernel & inference workflow
SWE-bench [Jimenez et al., 2024]	✓	✓	–	–	Issue resolution (Python)
SWE-bench-Live [Zhang et al., 2025]	✓	✓	–	–	Live issue resolution (Python)
SWE-Perf [He et al., 2026]	✓	✓	–	–	Performance optimization (Python)
GSO [Shetty et al., 2025]	✓	✓	✓	–	Performance optimization (CPU)
ParEval-Repo [Davis et al., 2025]	✓	–	✓	✓	Parallel model translation
ISO-Bench [Nangia et al., 2026]	✓	✓	–	✓	Inference engine optimization
SWE-GPU-Bench (ours)	✓	✓	✓	✓	**Bug fix, feature, performance**

GPU kernel benchmarks. KernelBench, TritonBench, CUDABench, MultiKernelBench, and ComputeEval evaluate generated CUDA or Triton kernels for functional correctness and runtime performance [Ouyang et al., 2025a, Li et al., 2025a, Zhu et al., 2026, Wen et al., 2025, NVIDIA, 2025]. TritonGym, robust-kbench, GEAK, FlashInfer-Bench, and SOL-ExecBench incorporate agentic refinement, inference workloads, or hardware-limit targets [Guan et al., 2026, Lange et al., 2025, Wang et al., 2025, Xing et al., 2026, Lin et al., 2026]. These benchmarks evaluate individual kernels or operators in isolation, abstracting away the host bindings, build systems, and cross-file dependencies required to integrate changes into a full repository.

Repository-level software engineering benchmarks. RepoBench, CrossCodeEval, and ReCUBE evaluate code completion with repository context [Liu et al., 2024, Ding et al., 2023, Hong et al., 2026],

while work on repository-level reasoning highlights the difficulty of cross-file retrieval [Ouyang et al., 2025b]. SWE-bench, SWE-Perf, FEA-Bench, OmniCode, and Rust-SWE-bench evaluate issue resolution, performance optimization, and feature implementation in real repositories [Jimenez et al., 2024, He et al., 2026, Li et al., 2025b, Sonwane et al., 2026, Xiang et al., 2026], commonly using agent frameworks such as SWE-agent or localization pipelines such as Agentless [Yang et al., 2024, Xia et al., 2025]. To overcome static dataset contamination and manual environment curation in SWE-bench, SWE-bench-Live [Zhang et al., 2025] introduces a continuously updated Python issue-resolution benchmark built with RepoLaunch, an agentic pipeline that automatically provisions per-instance containers and validates fail-to-pass tests. While SWE-GPU-Bench similarly uses a coding agent to synthesize and replay container-level setup and test commands from real PRs, it targets the GPU software stack rather than CPU-bound Python packages—requiring multi-language source builds (CUDA, C++, Python, and CMake/Bazel) and architecture-specific execution on GPU hardware—and extends evaluation beyond bug fixes to feature implementations and correctness-gated performance optimizations. Similarly, GSO includes repositories that contain GPU code, but its released tasks execute on CPUs [Shetty et al., 2025], whereas SWE-GPU-Bench compiles and validates patches on GPUs.

Repository-level GPU benchmarks. Table 4 compares SWE-GPU-Bench with representative kernel-level and repository-level benchmarks. ParEval-Repo evaluates code translation of HPC applications across parallel programming models (CUDA, OpenMP, and Kokkos), identifying build systems and cross-file dependencies as primary bottlenecks [Davis et al., 2025]. ISO-Bench evaluates performance optimization commits drawn from LLM inference engines (vLLM and SGLang) [Nangia et al., 2026]. Complementing these efforts, SWE-GPU-Bench targets everyday PR-driven maintenance across diverse layers of the GPU software stack, combining multi-language repository edits and source-level GPU builds across bug fixes, feature implementations, and correctness-gated performance optimizations.

6 Discussion

Limitations and future directions. Several limitations shape the scope of our findings. First, because the setup commands S are executed before the agent starts, SWE-GPU-Bench decouples code maintenance from environment provisioning; evaluating whether agents can autonomously configure GPU toolchains and resolve system dependencies from scratch is an important direction for future work. Second, to ensure isolatable fail-to-pass test oracles, our filtering pipeline bounds patch size and excludes broad architectural refactors and release merges, scoping the benchmark to PR-scale maintenance. Third, all tasks are validated on NVIDIA A100 and H100 GPUs and have not yet been validated on newer architectures such as Blackwell or non-NVIDIA accelerators; tasks targeting specific compute capabilities (such as `sm_80` or `sm_90a`) and all runtime measurements require re-validation when ported to other hardware. Fourth, standardized performance evaluation across heterogeneous GPU workloads remains an open challenge: our Perf-Replay subset relies on task-specific benchmark commands that output scalar metrics on a relatively small set of tasks, supporting preliminary observations rather than definitive speedup rankings. Finally, problem statements are distilled by an LLM prior to manual audit, and each model configuration is evaluated in a single pass@1 run; Appendix D discusses additional considerations including pretraining contamination, test adequacy, and broader impacts.

Conclusion. We introduced SWE-GPU-Bench, a benchmark for evaluating language models and coding agents on repository-level GPU software engineering across CUDA, C++, Python, and heterogeneous build systems. By pairing real-world pull requests with container-verified setup, test, and performance benchmark commands, SWE-GPU-Bench measures end-to-end capabilities from cross-file localization and compilation to functional verification and correctness-gated speedup evaluation. While recent frontier models have advanced rapidly to reach nearly 100% compilation accuracy—overcoming the build failures that hindered earlier generations—the strongest agent configurations resolve up to 58.3% of tasks under our pass@1 evaluation protocol, leaving functional test verification and the correctness gate on performance tasks as major bottlenecks. We release the benchmark dataset, construction and evaluation pipeline, and container specifications to support future research on GPU coding agents.

Acknowledgments and Disclosure of Funding

The authors received no specific third-party funding for this work and declare no competing financial interests.

References

Joshua H. Davis, Daniel Nichols, Ishan Khillan, and Abhinav Bhatele. ParEval-Repo: A benchmark suite for evaluating LLMs with repository-level HPC translation tasks. In *Proceedings of the 54th International Conference on Parallel Processing*, pages 94–103. Association for Computing Machinery, 2025. URL https://doi.org/10.1145/3754598.3754669.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)*, volume 36, pages 46701–46723. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/920f2dced7d32ab2ba2f1970bc306af6-Abstract-Datasets_and_Benchmarks.html.

Yue Guan, Yichen Lin, Xu Zhao, Jianzhu Yao, Xinwei Qiang, Zhongkai Yu, Pramod Viswanath, Yufei Ding, and Adnan Aziz. TritonGym: A benchmark for agentic LLM workflows in Triton GPU code generation. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 37457–37473. PMLR, 2026. URL https://proceedings.mlr.press/v306/guan26d.html.

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 *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 41651–41666. PMLR, 2026. URL https://proceedings.mlr.press/v306/he26y.html.

Jiseung Hong, Benjamin G. Ascoli, and Jinho D. Choi. ReCUBE: Evaluating repository-level context utilization in code generation. *arXiv preprint arXiv:2603.25770*, 2026.

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

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

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 *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23053–23066. Association for Computational Linguistics, 2025a. URL https://aclanthology.org/2025.findings-acl.1183/.

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 *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 17160–17176. Association for Computational Linguistics, 2025b. URL https://aclanthology.org/2025.acl-long.839/.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, Dheeraj Peri, Ouye Xie, Aditya Kane, Moshe Maor, Michael Behar, Triston Cao, Rishabh Mehta, Vartika Singh, Vikram Sharma Mailthody,

Terry Chen, Zihao Ye, Hanfeng Chen, Tianqi Chen, Vinod Grover, Wei Chen, Wei Liu, Eric Chung, Luis Ceze, Roger Bringmann, Cyril Zeller, Michael Lightstone, Christos Kozyrakis, and Humphrey Shi. SOL-ExecBench: Speed-of-light benchmarking for real-world GPU kernels against hardware limits. In *Advances in Neural Information Processing Systems*, volume 39, 2026. URL https://openreview.net/forum?id=TE8B0CjNsw.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/d191ba4c8923ed8fd8935b7c98658b5f-Abstract-Conference.html.

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

NVIDIA. ComputeEval: Evaluating large language models for CUDA code generation. https://github.com/NVIDIA/compute-eval, 2025. Accessed: 2026-10-20.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 47356–47415. PMLR, 2025a. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/4a4a3c197deac042461c677219efd36c-Abstract-Conference.html.

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)*, volume 38, pages 167761–167804. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/ddd98389004161943e646322a16348c3-Abstract-Datasets_and_Benchmarks_Track.html.

Atharv Sonwane, Eng-Shen Tu, Wei-Chung Lu, Claas Beger, Carter Larsen, Debjit Dhar, Simon Alford, Rachel Chen, Ronit Pattanayak, Tuan Anh Dang, Guohao Chen, Gloria Geng, Kevin Ellis, and Saikat Dutta. OmniCode: A benchmark for evaluating software development agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 40634–40661. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.findings-acl.2020/.

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. *arXiv preprint arXiv:2507.23194*, 2025.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. MultiKernelBench: A multi-platform benchmark for kernel generation. *arXiv preprint arXiv:2507.17773*, 2025.

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.

Jiahong Xiang, Wenxiao He, Xihua Wang, Hongliang Tian, and Yuqun Zhang. Evaluating and improving automated repository-level Rust issue resolution with LLM-based agents. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pages 805–817. Association for Computing Machinery, 2026. URL https://doi.org/10.1145/3744916.3773108.

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 *Proceedings of Machine Learning and Systems*, volume 8, 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*, volume 37, pages 50528–50652. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.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)*, volume 38, pages 163347–163372. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/d83c4a745789690f82e86d0ef752ae7c-Abstract-Datasets_and_Benchmarks_Track.html.

Jiace Zhu, Wentao Chen, Qi Fan, Zhixing Ren, Junying Wu, Xing Zhe Chai, Chotiwit Rungrueangwutthinon, Yehan Ma, and An Zou. CUDABench: Benchmarking LLMs for text-to-CUDA generation. *arXiv preprint arXiv:2603.02236*, 2026.

A Benchmark Details

This appendix gives the details of the construction pipeline in Section 2.2.

A.1 Collection and Filtering

Repositories and PRs. We identify candidate repositories through GPU-related GitHub searches and CUDA-related repository signals, and retain those actively maintained by an official organization or by core maintainers. For each repository, we collect its merged PRs with their title, description, metadata, changed files, diff, and any available CI information.

Rule-based filter. A PR passes this filter if it modifies at least one test file and at least one GPU-relevant implementation file (CUDA, C++, Triton, ROCm, GPU runtime or backend code, kernel dispatch, or build and compilation files). We exclude PRs that modify only documentation, tests, release metadata, or pure Python code, and bound the number of modified files and lines across the full textual diff (distribution in Table 6). Bounding diff size excludes release merges and broad refactors that bundle multiple unrelated changes and cannot be reduced to a single task with an isolatable fail-to-pass test; SWE-GPU-Bench therefore targets PR-scale maintenance and leaves release engineering and architectural migrations out of scope.

LLM judge. The judge is `gpt-5.2-2025-12-11`, which we query in a single chat call without an agent harness. It places each candidate PR in one primary category (bug fix, performance, feature, compatibility, observability, build infrastructure, refactor, test-only, documentation-only, release, or mixed) and rates its GPU relevance, problem clarity, test linkage, and reproducibility on A100 or H100 GPUs. A PR advances only if the judge accepts it or marks it as borderline, it changes core code, and it is rated reproducible. Its GPU relevance and problem clarity must also not be rated low, and its tests must be strongly linked to the change, or moderately linked with a clear oracle. Before freezing the benchmark, we refined the judge prompt on observed failure cases. Appendix E.1 describes the prompt.

Verification and final composition. Candidates passing the rule-based and LLM filters proceed to interactive command synthesis (Stage III), clean fail-to-pass replay in an agent-free container (Stage IV), and manual audit (Stage V), yielding 417 verified tasks across 30 repositories: 378 general-set tasks and 39 Perf-Replay tasks (spanning 399 unique pull requests, with 18 instances shared across both tracks).

A.2 Task Format

Each task is a single JSON record with the fields in Table 5. The listing below shows the problem statement of CUTLASS PR #1275, whose synthesized commands appear in Appendix A.5.

```
Extend the GEMM epilogue "tensor broadcast" bias feature to support broadcasting a 1D bias vector either
     per output row (existing behavior) or per output column, selectable at compile time. Preserve the
     current default behavior so existing code and kernels continue to treat the bias as per-row unless
     explicitly opted into per-column mode.

Also ensure the host-side reference epilogue computation used by utilities/tests applies the bias
     consistently with the selected broadcast direction, so correctness checks match device behavior.

Constraints:
- Do not modify, add, or update any test sources or golden outputs.
- Keep public APIs/backward compatibility: existing template instantiations and callers must compile
     unchanged.
- The per-column option must work with the existing combination of scaling (alpha/beta), bias binary op,
     and activation/unary/binary epilogue features.

Success criteria:
- Repository builds successfully with CUDA targets enabled.
- Existing unit tests pass unchanged.
- A kernel configured to use per-column bias produces correct results (matching the host reference)
     across typical MxN shapes and batch/L dimensions.
```

Table 5: Fields of a SWE-GPU-Bench task.

Field	Description
`instance_id`	Unique identifier, `owner__repo__PR`.
`repo`, `pr_number`	Source repository and pull request number.
`base_commit`, `head_commit`	Commit before the pull request and its merged head.
`problem_statement`	Task description shown to the model.
`task_type`	Bug fix, feature, or performance.
`setup_commands`	Commands that provision the environment.
`test_commands`	Commands that compile and run the tests of the pull request.
`patch`	Gold patch Δ_{gold} (hidden from the model).
`test_patch`	Test patch Δ_{test} (hidden from the model).
`perf_commands`	Performance commands P (Perf-Replay tasks only).
`docker_image`	Base container image.

A.3 Size Distributions

Patch sizes are heavy-tailed (Table 6): the median gold patch changes 80 lines in 3 files, while the largest gold patches reach 3,227 lines and 62 files.

Table 6: Distribution of size attributes over all tasks.

Attribute	Mean	SD	P25	P50	P75	P90	P95	P99	Max
Gold-patch lines	178.6	308.6	21	80	200	415.2	631.8	1,339.9	3,227
Test-patch lines	126.0	148.9	33.5	76	155.5	328.6	433.6	705.2	927
Gold-patch files	3.7	4.4	1	3	5	7	9	19	62
Problem words	236.6	237.2	132	190	261	427.2	621.2	1,347.2	2,636

A.4 Manual Review

Every candidate task that passes replay validation is manually audited.

Problem statement. The problem statement must describe the intended change accurately without revealing implementation details from the gold patch. An automated script first flags tokens in the statement that also appear in the gold patch, including file paths, function names, internal identifiers, and code fragments. Reviewers inspect each flagged token—retaining public API names required by the tests or stated in the original issue—and revise or reject any statement that discloses edit locations, introduces unsupported requirements, omits core requirements, or requests test modifications.

Test commands. Although replay validation ensures that T fails before applying the gold patch and passes afterward, reviewers additionally verify that T directly exercises the test files, functions, markers, or binaries introduced in Δ_{test}. Commands that miss the modified tests or execute an unnecessarily broad test suite are revised or rejected.

Performance commands. For Perf-Replay tasks, reviewers verify that P executes the benchmark relevant to the PR and that the evaluation targets the primary metric (latency, throughput, or kernel time) corresponding to the PR's optimization goal.

A.5 Example: Synthesized Commands for CUTLASS PR #1275

CUTLASS PR #1275 ("Allow per-column bias in EpilogueTensorBroadcast", +71/−9 lines across 4 files) adds a compile-time per-column bias option alongside an SM90 GoogleTest unit test. Because the PR description does not specify build or test instructions, the command-synthesis agent located the CUDA compiler, configured CMake for the `sm_90a` target architecture, compiled the CUTLASS test harness and the modified test source, linked against the CUDA runtime and driver stubs, and filtered execution to the new GoogleTest case. In the listing below, highlighted fragments depend on the environment or target architecture, and `$INC` abbreviates repeated include paths.

```
# setup: configure the build for SM90a
export CUDACXX=/usr/local/cuda/bin/nvcc
cmake -S repo -B artifacts/build -DCUTLASS_NVCC_ARCHS=90a -DCUTLASS_ENABLE_TESTS=ON \
      -DCUTLASS_ENABLE_CUBLAS=OFF -DCUTLASS_ENABLE_CUDNN=OFF
# setup: compile GoogleTest and the CUTLASS test harness
c++ -O3 -std=c++17 $INC -isystem /usr/local/cuda/include -c .../gtest-all.cc
c++ -O3 -std=c++17 -DCUTLASS_TEST_LEVEL=0 $INC -c repo/test/unit/common/filter_architecture.cpp
c++ -O3 -std=c++17 -DCUTLASS_TEST_LEVEL=0 $INC -c repo/test/unit/test_unit.cpp
# test: compile the changed test for SM90a
/usr/local/cuda/bin/nvcc -O3 -std=c++17 --generate-code=arch=compute_90a,code=[sm_90a] \
      -DCUTLASS_ENABLE_TENSOR_CORE_MMA=1 --expt-relaxed-constexpr $INC \
      -c repo/test/unit/gemm/device/sm90_gemm_f32_f32_f32_tensor_op_f32_tensor_broadcast.cu
# test: link against the CUDA runtime and driver stubs
c++ *.o -o artifacts/manual_build/percol_bias_test -L/usr/local/cuda/lib64/stubs \
      /usr/local/cuda/lib64/libcudart.so /usr/local/cuda/lib64/stubs/libcuda.so -lpthread -ldl
# test: run only the new test case
./artifacts/manual_build/percol_bias_test \
  --gtest_filter=SM90_Device_Gemm_f32t_f32n_f32n_tensor_op_gmma_f32_tensor_broadcast.*_PerColBias
```

B Experimental Details

B.1 Models and Harnesses

Table 7 lists the evaluated models and their versions. In the oracle setting, each model is queried once with the prompt in Appendix E.4. In the agent setting, GPT models run in Codex CLI and the other models in OpenCode, with the prompt in Appendix E.5.

Dataset revision and evaluation denominators. During the camera-ready revision, we audited and updated the benchmark dataset by removing invalid tasks and refining task definitions and verification scripts. Recent frontier models—specifically GPT-5.6-Sol and subsequent models (GPT-6-Astra, GPT-6.1-Sol, DeepSeek-V4.1-Flash, and DeepSeek-V4-Pro)—were evaluated across the full updated general benchmark set. Because the API endpoints for several earlier models were no longer accessible at the time of the revision, these earlier baselines could not be re-run on newly added or updated tasks; their metrics in Table 2 (and the fine-grained breakdowns in Section 4 and Appendix C) are therefore computed over the retained subset of valid tasks from their prior evaluation runs, resulting in different evaluation denominators across model generations.

Table 7: Evaluated models. A dash in the harness column marks a model run only in the oracle setting.

Model	Version or endpoint	Settings	Agent harness
GPT-6.1-Sol	`gpt-6.1-sol`	Oracle, agent	Codex CLI
GPT-6-Astra	`gpt-6-astra`	Oracle, agent	Codex CLI
GPT-5.6-Sol	`gpt-5.6-sol`	Oracle, agent	Codex CLI
GPT-5.5	`gpt-5.5`	Agent	Codex CLI
GPT-5.2	`gpt-5.2-2025-12-11`	Oracle	–
GPT-5-Codex	`gpt-5-codex`	Oracle, agent	Codex CLI
GPT-5.1-Codex	`gpt-5.1-codex-2025-11-13`	Oracle, agent	Codex CLI
GPT-5 mini	`gpt-5-mini-2025-08-07`	Agent	Codex CLI
Claude Sonnet 4	`claude-sonnet-4`	Oracle	–
Gemini 2.5 Pro	`gemini-2.5-pro-preview-05-06`	Oracle	–
GLM-5	`glm-5`	Oracle	–
Kimi K2.5	`kimi-k2.5`	Oracle, agent	OpenCode
DeepSeek-V4.1-Flash	`ds-v4.1-flash`	Oracle, agent	OpenCode
DeepSeek-V4-Pro	`ds-v4-pro`	Oracle, agent	OpenCode
DeepSeek-V4-Flash	`ds-v4-flash`	Oracle, agent	OpenCode
DeepSeek-V3.1	`deepseek-v3.1`	Oracle, agent	OpenCode
DeepSeek-V3.2	`deepseek-reasoner`	Oracle, agent	OpenCode
DeepSeek-R1	`deepseek-r1`	Oracle	–

B.2 Cost

Command synthesis. Table 8 summarizes the token usage and runtime of command synthesis with GPT-5.1-Codex for both setup-and-test commands and performance commands across candidate repositories.

Table 8: Tokens and runtime per command-synthesis run.

Run type	Runs	Mean	Median	P90	P95	Max	Mean time
Setup and test commands	1,259	4.83M	3.46M	10.14M	13.83M	32.35M	17.0 min
+ performance commands	292	4.00M	2.55M	8.53M	13.53M	29.08M	14.4 min

Model evaluation. Table 9 reports the per-task token consumption and wall-clock runtime across both the agent and oracle settings for the five most recent frontier models on A100 and H100 GPUs (wall-clock runtime depends on API latency and serving load, so it reflects empirical measurements in our evaluation environment):

- **Multi-turn agent interaction consumes $\sim 64\times$ more tokens per task than single-pass oracle generation.** In the agent setting, tasks average $3{,}018{,}335$ total tokens ($2{,}988{,}657$ input and $29{,}678$ output tokens), compared with $46{,}937$ total tokens ($42{,}182$ input and $4{,}755$ output tokens) in the oracle setting, reflecting repeated context accumulation from repository traversal, build logs, and test feedback.
- **Container compilation and test grading account for 4–5 minutes per task in both settings.** In the oracle setting, single-pass model inference averages 1.14 min out of 5.72 min total task time, with the remaining ~ 4.6 min spent on container setup, compilation, and test execution. In the agent setting, total task duration averages 20.81 min, of which 16.51 min is spent in the interactive agent execution stage and ~ 4.3 min in final patch verification.
- **Frontier models differ substantially in token and time efficiency.** Among agent configurations, GPT-6-Astra and GPT-6.1-Sol achieve the highest resolution rates (58.3% and 58.0%) with the lowest token footprints (2.43M and 1.71M total tokens per task) and shortest model execution times (9.55 and 6.50 min). By contrast, DeepSeek-V4-Pro generates $61{,}174$ output tokens per task on average—more than four times the output volume of the GPT agents—resulting in a mean model execution time of 35.18 min (39.36 min total) while resolving 43.3% of tasks.

Table 9: Per-task mean runtime and token usage for recent frontier models in the agent and oracle settings. **Total time** is the end-to-end task duration (including container grading), and **Model time** is the model/agent execution stage.

Setting	Model	Total time (min)	Model time (min)	Input tokens	Output tokens	Total tokens
Agent	GPT-6.1-Sol	11.26	6.50	1,699,679	12,932	1,712,611
	GPT-6-Astra	14.26	9.55	2,413,145	13,848	2,426,993
	GPT-5.6-Sol	15.66	11.00	3,480,624	14,327	3,494,950
	DeepSeek-V4.1-Flash	16.31	12.80	3,589,341	32,996	3,622,337
	DeepSeek-V4-Pro	39.36	35.18	2,968,245	61,174	3,029,418
	Mean	20.81	16.51	2,988,657	29,678	3,018,335
Oracle	GPT-6.1-Sol	5.83	1.28	43,811	4,634	48,446
	GPT-6-Astra	6.09	1.86	43,139	5,607	48,746
	GPT-5.6-Sol	6.26	1.15	42,674	5,603	48,277
	DeepSeek-V4.1-Flash	5.12	0.31	39,167	7,222	46,389
	DeepSeek-V4-Pro	4.78	0.46	39,898	1,786	41,684
	Mean	5.72	1.14	42,182	4,755	46,937

C Additional Results

Table 10 lists the values plotted in Figures 5–7 for the four representative baseline models (computed over their prior evaluation runs as noted in Appendix B.1).

Table 10: Resolution rate (%) by task type and by the number of files and languages edited by the gold patch. Perf.: performance.

		Task type			Files edited			Languages edited			
Setting	Model	Bug fix	Feature	Perf.	1–2	3–4	5+	1	2	3	4+
Oracle	GPT-5.1-Codex	25.8	8.6	5.4	24.6	15.9	4.8	26.7	10.6	5.4	10.0
Oracle	DeepSeek-V3.1	17.8	7.7	10.8	17.7	14.0	4.0	19.4	8.9	6.5	3.3
Oracle	DeepSeek-V3.2	16.5	4.0	0.0	15.7	7.0	3.4	17.0	4.2	4.5	4.0
Oracle	Kimi K2.5	23.8	14.4	9.4	23.2	22.2	9.9	25.2	15.2	14.5	7.7
Agent	GPT-5.1-Codex	37.3	25.6	16.7	37.5	27.3	21.4	42.0	25.0	18.2	12.5
Agent	DeepSeek-V3.1	31.0	17.2	18.9	29.3	20.8	17.9	30.0	19.2	15.7	28.6
Agent	DeepSeek-V3.2	28.9	23.1	16.7	27.8	45.0	17.1	32.4	30.0	26.9	11.8
Agent	Kimi K2.5	34.0	25.6	16.7	31.6	40.0	17.1	36.1	35.0	19.2	11.8

C.1 Perf-Replay Case Studies

Because performance tasks span heterogeneous metrics, workload shapes, and hardware bottlenecks, aggregate speedup numbers cannot be interpreted as a direct cross-model ranking. Even when a candidate patch passes all functional tests in T, its runtime behavior depends on low-level implementation choices relative to the reference gold patch. Table 11 reports the normalized task-level performance gains for two representative agents in Table 3 (GPT-5.1-Codex and Kimi K2.5), and Figures 8–10 contrast the generated code against the gold patch on three representative tasks.

Table 11: Normalized task-level performance gains on representative Perf-Replay tasks passing functional tests, aligned with **Perf. gain (%)** in Table 3.

Model	Task	Metric	Agent gain (%)	Gold gain (%)
GPT-5.1-Codex	`cutlass#53`	Throughput	0.3	0.1
	`flashinfer#588`	Latency	2.1	11.4
	`tensorflow#113198`	Throughput	7.5	7.7
Kimi K2.5	`cuda-python#459`	Latency	6.5	6.3
	`cutlass#53`	Throughput	0.1	0.1

Case 1: Kernel launch configuration and host–device synchronization (`flashinfer#588`). This task replaces a batch-parallel paged KV-cache append kernel (`dim3 nblks(batch_size)`, which under-utilizes the GPU at small batch sizes) with an element-parallel implementation. Both the gold patch and GPT-5.1-Codex convert ragged `append_indptr` offsets into per-token `(batch_indices, positions)` metadata and pass all functional tests, but their implementations differ in three key places (Figure 8): the gold patch precomputes metadata outside the timed loop, bounds grid size via occupancy query, and uses fast reciprocal division (`uint_fastdiv`), whereas GPT-5.1-Codex recomputes metadata inside the timed region, triggers an implicit device-to-host synchronization via `.item<int32_t>()`, and launches one block per token with standard integer division.

Case 2: Bit-packed tagged pointers vs. double-word atomics (`tensorflow#113198`). This task eliminates spin-waiting in XLA's `ObjectPool` by replacing a mark-bit spin lock with lock-free `PopEntry` and `PushEntry` operations while preventing ABA pointer reuse (Figure 9). The gold patch aligns each pool entry to a 64-byte cache line to avoid false sharing and packs a 13-bit ABA version tag into unused bits of a single 64-bit `uintptr_t`, whereas GPT-5.1-Codex leaves `Entry` unaligned and pairs the pointer with a full 64-bit counter in a 16-byte `Head` struct requiring 128-bit double-word CAS.

Gold Patch (Δ_{gold}: +11.4% gain)

```
// [1] Precompute metadata outside timed loop
batch_idx, pos = get_batch_indices_positions(...);

// [2] Occupancy-bounded grid-stride launch
cudaOccupancyMaxActiveBlocksPerMultiprocessor(
    &b_per_sm, kernel, num_threads, 0);
int n_cta = min(b_per_sm, ceil_div(nnz, sms)) *
     sms;
dim3 nblks(n_cta);

// [3] Fast reciprocal divmod in kernel loop
for (uint32_t i = cta_id; i < nnz; i += num_ctas)
     {
  paged_kv.page_size.divmod(
      indptr[batch_idx[i]] * page_sz + pos[i],
      page_iter, entry_idx);
}
```

GPT-5.1-Codex Patch ($\hat{\Delta}$: +2.1% gain)

```
// [1] Recompute in timed loop + implicit D2H sync
int32_t total = append_indptr[-1].item<int32_t>();

// [2] Unbounded launch: 1 CTA per token
//     (no occupancy query or grid-stride loop)
uint32_t num_tokens = append_key.size(0);
dim3 nblks(num_tokens);

// [3] Standard integer division/modulo in kernel
uint32_t i = blockIdx.x;
if (i < num_tokens) {
  page_iter = indptr[batch_idx[i]]
              + seq_pos[i] / page_size;
  entry_idx = seq_pos[i] % page_size;
}
```

Figure 8: Implementation comparison on `flashinfer-ai/flashinfer#588`: (1) precomputed metadata vs. per-call device-to-host synchronization (`.item()`), (2) occupancy-bounded grid-stride loop vs. one CTA per token, and (3) fast reciprocal `divmod` vs. integer division.

Gold Patch (Δ_{gold}: +7.7% gain)

```
// [1] 64B cache-line aligned entry (no false
     sharing)
struct alignas(64) Entry {
  std::optional<T> object;
  Entry* next = nullptr;
};
// [2] Pack 13-bit ABA tag into 64-bit pointer
     word
//     Layout: [7 hi tag bits | ptr | 6 lo tag
     bits]
std::atomic<uintptr_t> head_{0};

// [3] Single-word 64-bit atomic CAS
while (Entry* ptr = GetPtr(head)) {
  uintptr_t next = MakeTagged(ptr->next, head);
  if (head_.compare_exchange_weak(head, next, ...)
     )
    return std::unique_ptr<Entry>(ptr);
}
```

GPT-5.1-Codex Patch ($\hat{\Delta}$: +7.5% gain)

```
// [1] Unaligned entry (entries share cache lines)
struct Entry {
  std::optional<T> object;
  Entry* next = nullptr;
};
// [2] Separate 64-bit counter in 16-byte struct
struct alignas(16) Head { Entry* ptr; uintptr_t tag
     ; };
std::atomic<Head> head_{Head{nullptr, 0}};

// [3] Double-word 128-bit atomic CAS
while (head.ptr != nullptr) {
  Head next{head.ptr->next, head.tag + 1};
  if (head_.compare_exchange_weak(head, next, ...)
     )
    return std::unique_ptr<Entry>(head.ptr);
}
```

Figure 9: Implementation comparison on `tensorflow/tensorflow#113198`: (1) 64-byte cache-line alignment vs. default alignment, (2) 13-bit version tag packed into a 64-bit pointer vs. a 16-byte `Head` struct, and (3) 64-bit single-word CAS vs. 128-bit double-word CAS.

Case 3: Constructor-time vs. loop-level pointer disambiguation (`cutlass#53`). In CUTLASS's Volta Tensor Core GEMM store iterator, indexing `pointer[ptr_idx]` with a thread-dependent offset inside the unrolled store loop prevents NVCC from keeping the shared-memory pointers in registers (Figure 10). Whereas the gold patch folds the warp-parity check into the constructor by swapping `pointer[0]` and `pointer[1]` once so that `store()` uses a compile-time constant index, both agents leave the constructor unchanged and instead eliminate dynamic array indexing inside `store()` via a precomputed register array (GPT-5.1-Codex) or a ternary select (Kimi K2.5).

D Limitations and Broader Impact

Limitations. In addition to the limitations discussed in Section 6, SWE-GPU-Bench is drawn from a finite set of open-source repositories and does not span every GPU software stack or accelerator family. We exclude tasks that require non-NVIDIA accelerators at runtime, private infrastructure, or external services. Because tasks are constructed from public GitHub PRs, some repositories and commits may overlap with model pretraining corpora, and we have not quantified pretraining contamination. Moreover, passing the PR-derived tests verifies the behavior covered by Δ_{test}, which may not exhaustively test all edge cases or downstream invariants of the repository.

Gold Patch (Δ_{gold}: $+0.1\%$ gain)

```
// [1] Swap pointers once in iterator constructor
if (((threadIdx.x >> 5) * Iterations::kD) & 2) {
  Scalar *tmp = pointer[0];
  pointer[0] = pointer[1]; pointer[1] = tmp;
}

// [2] Compile-time constant index in unrolled
      store()
CUTLASS_PRAGMA_UNROLL
for (int d = 0; d < Iterations::kD; ++d) {
  Scalar *_pointer = pointer[(d & 1) ^ ((d >> 1) &
       1)];
  Store<...>::store(..., _pointer, offset);
}
```

Agent Patches ($\hat{\Delta}$: $+0.3\%$ / $+0.1\%$ gain)

```
// [1] GPT-5.1-Codex: precompute register array in
       store()
Scalar *lane_ptrs[Iterations::kD];
for (int d = 0; d < Iterations::kD; ++d)
  lane_ptrs[d] = ptr_idx(d) ? pointer[1] : pointer
      [0];
Scalar *_pointer = lane_ptrs[d];

// [2] Kimi K2.5: hoist warp_id + ternary select
      in store()
int warp_id = (threadIdx.x >> 5);
int ldg_idx = d + warp_id * Iterations::kD;
int ptr_sel = ((ldg_idx & 1) ^ ((ldg_idx >> 1) &
      1));
Scalar *_pointer = (ptr_sel ? pointer[1] : pointer
      [0]);
// (both leave constructor unchanged)
```

Figure 10: Implementation comparison on `NVIDIA/cutlass#53`: the gold patch resolves warp-dependent pointer order once in the constructor so the unrolled `store()` loop uses compile-time constant indices, whereas the agents eliminate dynamic indexing locally inside `store()`.

Broader impact. SWE-GPU-Bench provides an empirical benchmark for evaluating and developing coding agents on GPU software, which underpins modern machine-learning training and inference infrastructure. Capable coding agents could reduce maintenance overhead across complex GPU codebases; however, automated edits to performance-critical kernels and runtime code risk introducing subtle numerical discrepancies or performance regressions, so human review and comprehensive testing remain essential before deployment. The benchmark is constructed exclusively from public open-source repositories, contains no personal information beyond public repository metadata, and executes candidate patches inside isolated containers.

E Prompts

This appendix documents the prompts used to construct and evaluate SWE-GPU-Bench. The Stage V problem-statement generation prompt (Appendix E.2) and the evaluation prompts for the oracle and agent settings (Appendices E.4 and E.5) are reproduced verbatim. Because the prompts for Stage II PR filtering (`prompts/fuse_filter.md`) and Stage III command synthesis (`prompts/agent_explore.md`, and `prompts/agent_perf_repro.md` for performance commands) are multi-page operational specifications, we summarize their criteria and output schemas below and include the full templates in the `prompts/` directory of the code release.

E.1 PR Filtering Prompt

The PR-filtering prompt specifies a rubric for the LLM judge to evaluate whether a merged PR is suitable as a GPU benchmark instance. Given the PR title, description, metadata, diff, modified file list, test diff, and available CI logs, the judge evaluates the candidate based solely on evidence present in these artifacts.

Required judgments. Each PR is assigned a primary category, optional secondary categories, and a scope level (`single_file`, `single_module`, `multi_module`, or `repo_level`). The rubric evaluates whether the PR modifies core GPU-relevant code, whether the accompanying tests exercise the modified code path, the strongest visible validation level, hardware reproducibility on A100 or H100 GPUs, and overall ratings for GPU relevance, problem clarity, test signal, and benchmark fit.

Categories. Allowed primary categories are `bugfix`, `perf`, `feature`, `compatibility`, `observability`, `build_infra`, `refactor`, `test_only`, `docs_only`, `chore_release`, and `mixed`. When a PR spans multiple aspects, the rubric prioritizes behavioral changes over maintenance categories according to the order `bugfix` $>$ `perf` $>$ `feature` $>$ `compatibility` $>$ `observability` $>$ `build_infra` $>$ `refactor` $>$ `test_only` $>$ `docs_only` $>$ `chore_release`.

Hardware reproducibility. The judge assesses whether the PR is reproducible on A100, H100, both, or neither (or marks `unclear` when evidence is insufficient). Candidates reproducible on A100 or H100 GPUs are retained, whereas PRs requiring unavailable hardware, non-NVIDIA runtime execution, or external services are rejected.

Benchmark-fit rule. Acceptance requires non-low GPU relevance and problem clarity, a substantive core-code modification, coupled test changes with strong test linkage (or medium linkage paired with a clear test oracle), hardware reproducibility on A100/H100, and a plausible local replay path. Candidates that satisfy most criteria but have a minor weakness on one dimension are labeled `borderline`; candidates lacking GPU relevance, core-code changes, test validation, or compatible hardware are labeled `reject`.

Output format. The judge outputs a JSON object conforming to the following schema:

```
{
  "pr_category": {
    "primary": "",
    "secondary": []
  },
  "core_code_change": false,
  "has_test_change": false,
  "test_link_strength": "",
  "code_test_coupled": false,
  "scope_level": "",
  "validation_level": "",
  "gpu_relevance": "",
  "problem_clarity": "",
  "test_signal": "",
  "hardware_reproducibility": "",
  "benchmark_fit": "",
  "reason": [
    "",
    "",
    ""
  ],
  "main_risk": "",
  "task_brief": ""
}
```

Each entry in `reason` must cite concrete evidence from the PR artifacts; when evidence is insufficient, the rubric requires outputting `borderline` or `reject`.

E.2 Problem Statement Generation Prompt

```
<query_generator>
You are a task distillation assistant.

Your job is to convert GitHub pull request metadata and code patch information into a single English task
      prompt for a coding model.

The coding model will be given the repository itself, but it must NOT be given implementation spoilers
     from the patch.

Your goal is to produce a prompt that describes:
1. what needs to be achieved,
2. what constraints must be respected,
3. what counts as success,

while hiding patch-specific solution details.

## Input
You will receive:
- PR title
- PR body
- optional issue/context text
- the required interface: names the PR introduces that the hidden tests call
- patch/diff

## Objective
Write one concise but clear English prompt that instructs a coding model to reproduce the intent of the
     PR in the repository.

## Hard constraints
Do NOT reveal or mention:
- file paths
```

```
- file names
- function names
- class names
- identifiers
- line numbers
- exact code snippets
- exact strings from the patch unless absolutely necessary for understanding the task
- specific edit strategy
- the sequence of modifications
- how many places were changed

Do NOT summarize the patch mechanically.
Do NOT say "modify X in file Y".
Do NOT expose the answer.

## Required interface exception
The hidden tests call every name listed under "Required interface" below. A coding model cannot guess
these names, so the prompt MUST mention each of them, as the public function, class, parameter or
binding that has to exist, and say briefly what it is for. This is the only exception to the
identifier rule above. Do not reveal how they are implemented. If the list says "(none)", mention
no identifiers.

## What the output prompt SHOULD contain
The prompt should:
- describe the user-visible or maintainer-visible goal of the change
- state any important scope restrictions
- preserve behavior unless the PR is intended to change behavior
- include acceptance criteria that can be checked from the repository state
- be phrased as an implementation task for a coding model

## Style requirements
- Output English only
- Output only the final prompt text, no explanation
- Keep it concrete, actionable, and repository-aware
- Prefer task intent over patch mechanics
- If the PR is small or localized, keep the prompt concise
- If the PR changes behavior, describe expected behavior and edge cases, but not the exact implementation

## Compression rule
Abstract from implementation details to task intent:
- patch detail -> behavioral/task requirement
- file-level evidence -> subsystem-level scope
- exact edits -> acceptance criteria

## Domain wording
Describe the change in the terms the PR itself supports. Many PRs in these repositories touch GPU
code (kernels, CUDA or HIP code paths, GPU-facing APIs, tensor/layout semantics, memory access or
synchronization, numerical correctness), but others are compiler passes, build logic, host-side
runtime or Python API changes. Use GPU wording only when the PR is actually about GPU behavior; do
not reframe other changes as GPU work, and do not invent performance, kernel, or numerical
requirements that the PR content does not support.

## Examples of good abstraction in GPU repositories
Bad: "Change the launch configuration in kernel_x to use 256 threads"
Good: "Correct the GPU execution configuration issue so that the affected code path behaves as intended
      under the expected workload, while preserving surrounding interfaces, numerical behavior, and
      compatibility constraints."

Bad: "Add a guard in the CUDA helper when stride == 0"
Good: "Fix the edge-case handling in the affected GPU utility logic so that invalid or degenerate inputs
      are handled correctly without introducing regressions in normal execution paths."

## Output checklist
Before producing the final prompt, ensure:
- it does not leak implementation details from the patch
- it names every entry of the required interface, and no other identifiers
- it is specific enough for a coding model to act on
- it reflects the actual intent of the PR
- it includes sensible constraints against unrelated edits

Very IMPORTANT:
Assume test patches are handled separately: the coding model must implement the task without modifying,
      adding, or updating any test files, fixtures, snapshots, or expected outputs.

Now read the PR information and generate the final prompt.

PR repo:
{repo}

PR title:
{title}
```

```
PR body:
{body}

Additional context:
{context}

Required interface:
{interface}

Patch:
{patch}
</query_generator>
```

E.3 Command Synthesis Prompt

The Stage III command-synthesis prompt (`prompts/agent_explore.md`) instructs the coding agent to construct reproducible setup and test commands (and `prompts/agent_perf_repro.md` analogously synthesizes performance benchmark commands). The prompt requires the agent to operate strictly as a reproducibility verifier—prohibiting source-code modifications, unverified test claims, or skipped steps—and to return a structured JSON verdict containing `setup_commands`, `test_commands`, and execution logs.

Objective and input. Given a JSON record containing repository metadata, PR number, title, body, base and head commit SHAs, diff text, and commit messages, the agent inspects the checked-out repository at `head_sha`, provisions the environment, identifies and executes the smallest test target that covers the PR's changes, and records the resulting command lists.

Environment and build policy. The agent inspects the repository's build scripts, CI configurations, and documentation to configure Python packages and native dependencies—including CUDA, NVCC, CMake, Ninja, GCC/Clang, NCCL, and LLVM—matching the target repository and GPU architecture.

Test-selection priority. Candidate test targets are prioritized as follows: (1) tests added or modified by the PR, (2) tests directly exercising the modified modules, (3) the closest CI test target covering the modified code path, and (4) a minimal file- or module-level test target. Running an unrelated full test suite is prohibited.

Hardware and system requirements. For PRs modifying CUDA or Triton kernels, C++/CUDA extensions, custom operators, compiler passes, or distributed runtimes, the agent verifies whether execution requires specific GPU hardware, CUDA toolkit versions, compute capabilities, or native libraries, and marks the candidate as blocked when required hardware or system dependencies are unavailable in the container.

E.4 Oracle Setting Prompt

```
<oracle>
You are evaluating a benchmark instance with a fixed oracle context.

Use only the provided task description and code context.
Do not assume access to any files beyond the supplied context.
Return a minimal patch that fixes the task.

You will be provided with a partial code base and a task statement
explaining a problem to resolve.
<task>
{{problem_statement}}
</task>

The code context below is split into file blocks:
- ‘[start of path]‘ and ‘[end of path]‘ mark the exact file path for each block.
- Lines are prefixed as ‘0001| source text‘ for reference only.
- The ‘0001|‘ prefixes are not part of the files. Do not include them in the patch.
- Use the reference line numbers only to locate the target code regions.

<code>
{{golden-patch modified files}}
</code>

Return exactly one JSON object and nothing else. Do not return a unified diff.
```

````
The JSON object must have this shape:

```json
{
  "edits": [
    {
      "path": "relative/path/to/file.py",
      "old": "exact existing text block to replace",
      "new": "replacement text block"
    }
  ]
}
```

Rules:
- `path` must be one of the provided file paths unless you are adding a new helper file.
- `old` must be copied exactly from the file content, without the `0001|` line prefixes.
- For an existing file, `old` must match exactly once in that file.
- To add a new file, use `"old": ""` and put the complete file content in `new`.
- Do not include code-context line prefixes such as `0001|` in `old` or `new`.
- Prefer the smallest set of edits that solves the task.

The system will apply your JSON edits locally and generate the final unified diff with `git diff`.
Response:

</oracle>
````

E.5 Agent Setting Prompt

```
<agent_eval>
You are a software engineer working on a GPU/HPC open-source project.
A task has been assigned to you. Read the description carefully and implement the required changes in the
      repository.

Focus on:
- Writing correct, working code
- Making minimal, focused changes that address the described requirements
- Building and verifying your changes locally before finishing

Guidelines:
- The workflow has already prepared the base environment before you start
- Reuse the existing environment under `artifacts/` instead of recreating it from scratch
- If you modify compiled code (C++, CUDA, etc.), rebuild and reinstall the package to verify correctness
- You may run the existing test suite at any point to check your work
- Keep all build artifacts, virtual environments, and caches inside the `artifacts/` directory

<task>
{{task}}
</task>
</agent_eval>
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the benchmark scope, size, construction, and evaluation settings, and these claims match Sections 2–4. Comparisons between the oracle and agent settings are stated as empirical measurements of our evaluation configurations.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 6 lists the main limitations (a prepared environment, validation on A100/H100 only, the small Perf-Replay subset, LLM-written problem statements, and single pass@1 runs), and Appendix D discusses further ones.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper has no theoretical results; the only notation defines a task (Section 2.1).

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Sections 2.1 and 3 define the task, the metrics, and the evaluation settings. Appendices A and B give construction details, model versions, and costs, and Appendix E gives the prompts. The data and code are public (footnote 2).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in

some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The benchmark is released at https://huggingface.co/datasets/Froze/SWE-GPU-Bench and the construction and evaluation code at https://github.com/chenfengshijie/SWE-GPU-Bench, with instructions for running the evaluation.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 3 specifies the oracle and agent settings and the evaluation protocol, and Appendix B.1 lists the evaluated models and harnesses. The paper involves no model training.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: We report pass@1 rates from a single run per configuration, as in SWE-bench and SWE-Perf, because repeated agent runs on GPU tasks are expensive (Appendix B.2). We do not report error bars, and Section 6 cautions against reading small differences between models.

   Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section 3 and Appendix B.2 report the GPU type (one A100 or H100 per task) and the token usage and wall-clock runtime of command synthesis and model evaluation.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The authors have reviewed the NeurIPS Code of Ethics. The research uses publicly available software-engineering artifacts, focuses on benchmark construction and model evaluation, and includes leakage checks, license safeguards, and isolated evaluation procedures for responsible release.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: Appendix D discusses the positive impact (better agents for maintaining GPU software) and the risk that agent patches introduce numerical or performance regressions, which is why such patches still need review and testing.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release a high-risk model or a sensitive scraped dataset. The benchmark uses only public open-source repositories, and model patches are evaluated in isolated containers (Appendix D).

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All source repositories are public open-source projects, and the release credits each one by recording the source repository of every task.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: SWE-GPU-Bench is documented in Section 2 and Appendix A (task format, construction, statistics, and manual review) and is released with a dataset card at `https://huggingface.co/datasets/Froze/SWE-GPU-Bench`.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The work does not involve crowdsourcing experiments or human-subject studies; the human validation described in the appendix is an author/internal expert audit of benchmark artifacts. Therefore participant instructions, screenshots, and compensation details are not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve human-subject experiments or participant data collection; the described human validation is an author/internal expert review of software benchmark artifacts. IRB or equivalent review is therefore not applicable.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are part of the method: an LLM judge filters pull requests, an LLM writes problem statements, a coding agent synthesizes commands, and the evaluated systems are LLMs (Sections 2.2 and 3, Appendix E). LLMs were also used to edit the text of the paper and to help visualize results; the authors checked all content.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.