SWE-KERNEL-BENCH: A LIVE BENCHMARK FOR GPU KERNEL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce SWE-KERNEL-BENCH, a live GPU kernel optimization benchmark that is periodically refreshed with new tasks derived from performance improvements merged into widely used open-source repositories. Its automated pipeline converts merged pull requests into tasks, each pairing an unoptimized source implementation with the corresponding maintainer-reviewed and shipped gold implementation. The benchmark evaluates whether coding agents can attain this performance target while preserving correctness. The first release contains 161 tasks from 10 production repositories and spans CUDA and Triton. Evaluating the benchmark with eight contemporary agentic systems shows that the best-performing system produces correct kernels for 91% of tasks, yet matches or exceeds gold performance on only 54%, revealing substantial headroom in real-world GPU kernel optimization.

1 INTRODUCTION

Optimizing GPU kernels demands expert reasoning about memory movement, parallelism, compiler behavior, and hardware-specific features (Yu et al., 2026; Ouyang et al., 2025; Dai et al., 2026). Coding agents could help automate this work, but evaluating whether such agents can provide real value requires benchmarks that are grounded in current production optimization problems, and the solutions shipped to address them.

Building such a benchmark is challenging. It must represent real engineering work, cover diverse and demanding tasks, and support evaluation within a particular deployment context, such as a target hardware platform or class of optimization. Most importantly, the problems worth optimizing and the kernel optimizations that deliver meaningful value continually change as libraries, compilers, workloads, and hardware evolve. At the same time, as coding agents advance, static task sets can saturate and lose their ability to distinguish among them. A useful evaluation must therefore incorporate new tasks over time, motivating recent work on live software benchmarks (Jain et al., 2025; Zhang et al., 2025; Badertdinov et al., 2025).

To meet these requirements, we introduce SWE-KERNEL-BENCH (Figure 1), a GPU kernel optimization benchmark constructed automatically from merged performance pull requests (PRs) in widely used open-source repositories. These PRs record concrete performance problems and the optimized implementations that maintainers reviewed, accepted, and shipped, grounding benchmark tasks in production engineering practice. Repository-scale mining makes the benchmark live: it produces diverse tasks at scale and supports continual refreshes as additional performance PRs become available. Moreover, selecting repositories, kernel-change categories, and target hardware allows the benchmark to be configured for a particular evaluation scope. Importantly, the automated construction pipeline accommodates heterogeneous changes, without requiring PRs to follow a predetermined structure or include dedicated tests.

The SWE-KERNEL-BENCH pipeline (§3) first mines merged PRs from selected repositories and filters them to identify GPU kernel performance changes suitable for benchmarking. For each selected PR, it extracts an unoptimized reference code, called the *source*, and retains the accepted optimized implementation as the *gold*. The pipeline also constructs a matched workload harness and packages the artifacts as a standalone task with a shared calling convention. An agent receives the source but not the gold, and is tasked with producing a solution that matches or exceeds the gold performance. Figure 2 shows an abridged example task.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

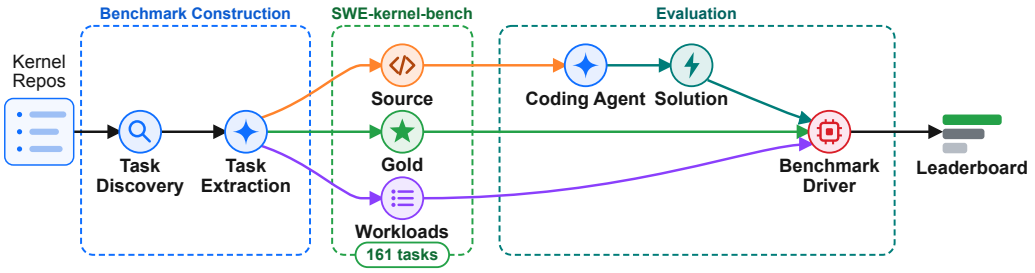


Figure 1: Evaluating coding agents with SWE-KERNEL-BENCH. Discovery mines performance optimizations from real-world engineering contributions, and extraction converts them into standalone tasks (§3). Each task comprises a *source* implementation, a *gold* implementation that defines the performance target, and a set of *workloads*. Coding agents optimize the source, and their solutions are benchmarked against the gold to produce the performance leaderboard (§5).

Source PyTorch

```python
def rotate_half(x):
    x1, x2 = x[..., :d//2], x[..., d//2:]
    return torch.cat((-x2, x1), dim=-1)

def apply_rotary_pos_emb(q, k, cos, sin):
    cos, sin = cos.unsqueeze(1), sin.unsqueeze(1)
    q_embed = (q*cos) + (rotate_half(q)*sin)
    k_embed = (k*cos) + (rotate_half(k)*sin)
    return q_embed, k_embed

def source_impl(*, q, k, cos_cache,
                sin_cache, pos_ids):
    cos = cos_cache[pos_ids]
    sin = sin_cache[pos_ids]
    return apply_rotary_pos_emb(q, k, cos, sin)
```

Gold CUDA

```cuda
__global__ void ApplyRopePosIdsCosSinCache(
    DType* q, DType* k, DType* q_rope, ...) {
  const IdType pos = pos_ids[idx];
  cos.load(cos_cache + pos*head_dim + tx*vs);
  sin.load(sin_cache + pos*head_dim + tx*vs);
  for (h = 0; h < num_qo_heads; ++h)
    q_vec = apply_rope_cos_sin(q_ptr,cos,sin);
  ...
}

def optimized_impl(*, q, k, cos_cache,
                   sin_cache, pos_ids):
    mod = _get_module()  # load_inline
    mod.apply_rope_pos_ids_cos_sin_cache(...)
    return q_rope, k_rope
```

Figure 2: A SWE-KERNEL-BENCH task (§3.1) includes the *source* task definition, the *gold* performance target, and the relevant *workloads* (see Appendix A.1). The source and gold codes use a shared calling convention. Agents solve tasks starting from the source alone.

Using this methodology, we construct our first benchmark release – a benchmark for kernel generation and optimization, comprising 161 tasks from 10 production libraries (§4), with gold implementations in both CUDA and Triton (Tillet et al., 2019). While the construction methodology is not tied to a particular architecture, the current benchmark focuses on tasks runnable on NVIDIA H100 GPUs. The shipped optimizations provide substantial headroom, outperforming their source implementations by a median of 2.19×.

We then evaluate eight leading models, each deployed within a coding-agent harness, in two settings: without GPU access and with compilation, correctness, timing, and profiling feedback (§5). The strongest configuration produces correct kernels on 91% of attempted tasks, but matches or beats the shipped gold performance on only 54%. Hardware feedback improves correctness for every evaluated system, yet the gap between correctness and matching gold shows that generating functional kernels does not imply reproducing production-level optimization.

Our contributions are as follows:

- **A live benchmark grounded in shipped GPU optimizations.** The first release of SWE-KERNEL-BENCH comprises 161 standalone tasks from 10 production libraries, including 88 CUDA and 72 Triton gold implementations. Each task pairs a real pre-optimization source with a maintainer-accepted gold that satisfies the correctness and performance admission criteria on the target hardware (§4).
- **A method for scalable benchmark construction from heterogeneous kernel-related repositories.** Our automated pipeline requires neither per-task manual construction nor pre-existing repository tests (§3).

- **An eight-system study of current kernel-optimization capabilities.** We evaluate eight leading coding systems on the first SWE-KERNEL-BENCH release (§5.3), finding that it leaves considerable headroom for improvement.

We will open-source this first release, the full construction pipeline, and the benchmarking harness. Subsequent releases will be published periodically and incorporate new tasks from the continuously growing pool of merged PRs.

2 RELATED WORK

Tasks from software history. SWE-bench (Jimenez et al., 2024) established repository history as a benchmark source: a repository state defines the task, and a merged change demonstrates solvability. Performance-oriented successors mine optimizations from general software, relying on repository tests, manual workloads, or tests generated from commit context (He et al., 2026; Ma et al., 2026; Shetty et al., 2025). A growing line of works extends this paradigm to *live* benchmarks, periodically refreshed from a stream of newly created, date-stamped instances rather than frozen at release (Jain et al., 2025; Zhang et al., 2025; Badertdinov et al., 2025; Huang et al., 2026a). SWE-KERNEL-BENCH specializes these approaches for GPU kernels, with a fully automated pipeline that extracts the changed computation and its shipped optimization into standalone programs, and automatically synthesizes correctness and performance workload tests.

Kernel benchmarks. A growing number of benchmarking works evaluate automated kernel generation capabilities. These works differ along multiple axes, including the provenance of the kernel tasks, the methods for curating the benchmark, and the performance targets that agents are measured against. Benchmarks like KernelBench (Ouyang et al., 2025) and TritonBench (Li et al., 2025) focus on closed sets of operators, exercising kernel generation at different levels; works like CUDAEval (Gong et al., 2026) opt for a more bottom-up approach, mining from large-scale kernel code data and relying on LLM-assisted extraction and categorization to automate the benchmark curation process.

More recently, some works have focused on benchmarking that is more grounded in real production behavior. FlashInfer-Bench (Xing et al., 2026) extracts kernel workloads by serving a set of popular LLMs against a dataset of user prompts, whereas Atrex-Bench (Yang et al., 2026) relies on automatic extraction from production inference traces that were collected over time. RealisticTritonBench (Huang et al., 2026b) is the closest work to ours in terms of task provenance, deriving kernel generation problems from merged PRs; their construction pipeline, however, involved substantial human effort, resulting in manual curation of 31 Triton tasks from roughly $2{,}000$ candidates.

Unlike these prior works, SWE-KERNEL-BENCH is a live, scalable and configurable benchmark for real shipped kernel optimizations in production code, relying on a fully automated pipeline that requires no human intervention. To our knowledge, SWE-KERNEL-BENCH is the first kernel-generation benchmark to combine real-world developer changes, fully automated task construction, and a continuously refreshable benchmark corpus.

3 BENCHMARK CONSTRUCTION

Benchmark construction consists of a two-stage pipeline. A *task discovery* phase mines performance PRs suitable for the benchmark. Then, *task extraction* distills each PR into one or more standalone, runnable tasks. We now detail the tasks structure and their construction.

3.1 TASK STRUCTURE

A task in SWE-KERNEL-BENCH consists of three artifacts extracted from a merged PR:

- The *source* is an unoptimized reference implementation that defines the task. It is constructed either from the pre-PR implementation, or from a reference implementation added by the PR.
- The *gold* is the optimized implementation shipped by the PR, setting the performance target.
- The *workload* defines the input shapes and data types used for evaluation, together with a random seeded input generator, grounded in the PR or surrounding repository code.

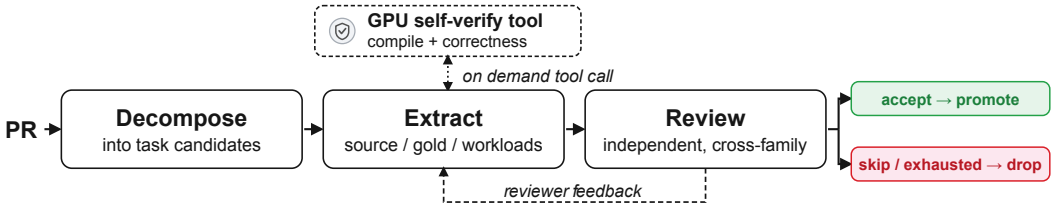


Figure 3: The SWE-KERNEL-BENCH extraction pipeline (§3.3). A decomposition agent partitions each PR into one or more task candidates. Each candidate then undergoes iterative extraction and review until it is accepted, skipped, or reaches the iteration limit.

The source and gold are standalone implementations with a shared call signature, allowing a generic driver to execute every task without installing the original repository or running its full application stack. Appendix A.1 presents an example benchmark task.

3.2 TASK DISCOVERY

The task discovery phase identifies performance-optimization PRs that can be converted into self-contained benchmark tasks. It operates as a data-mining funnel: inexpensive stages process many PRs, while progressively more compute-intensive stages evaluate increasingly selective subsets.

The funnel begins with merged PRs from open-source GPU projects, including serving engines, kernel libraries, compilers, and array frameworks (Appendix A.2). A rule-based filter removes empty, documentation-only, and configuration-only changes, then retains PRs containing GPU-related terms or structural signals in their metadata, file paths, or diffs. Next, a model (*gpt-oss-120b*) summarizes each PR and assigns it to one or more of the 11 kernel-change categories — or rejects it as irrelevant — using the definitions in Appendix A.3 and the prompt in Appendix B.1. This benchmark release targets two categories: `kernel-generate`, which introduces a GPU kernel, and `kernel-optimize`, which improves an existing kernel without changing its computation. Future live releases will include PRs from other categories as well.

A frontier model (*Claude Opus 5*) then reviews candidates carrying at least one of these categories, verifying that: (i) the PR indeed belongs to one of them, which the review settles by selecting the single best fit; (ii) a defensible before-and-after computation exists; (iii) the computation and synthetic inputs can be isolated; and (iv) the PR reports a speedup. The review also assigns each candidate an estimated difficulty and confidence score. Appendix B.1 provides the full gating prompt. Passing PRs proceed to task extraction.

The funnel is highly selective. From 55,210 merged PRs across 23 repositories, the deterministic filter retained 35,961 at negligible cost. The categorizer identified 5,655 as `kernel-generate` or `kernel-optimize`, and frontier review advanced 1,309 to extraction. This review is the only discovery stage using an expensive frontier model, costing $308 in total, or approximately $0.05 per candidate PR.

3.3 TASK EXTRACTION

Task extraction converts PRs that pass discovery into self-contained benchmark tasks. Given a PR URL, the procedure analyzes the change and its repository context to produce the source, gold, and workload harness files defined in §3.1, together with metadata and provenance records. Because a single PR may contain multiple independently benchmarkable optimizations, it can yield multiple task candidates. Candidates that cannot be extracted faithfully are excluded.

We implement extraction as a three-agent pipeline (Figure 3). First, a decomposition agent partitions each PR into independently benchmarkable task candidates. For each candidate, an extractor examines the repository before and after the change and constructs the source, gold, and workload harness files. Every file it writes must cite the repository path, revision, and line range it was derived from—the provenance records—and the extractor must refuse the candidate when no such origin exists rather than invent one. It may use a GPU self-verification tool to compile the source and gold

implementations and test their numerical equivalence on real hardware. A reviewer from a different model family then independently assesses whether the implementations faithfully represent the PR, their comparison is fair, the workloads are grounded in the PR, and the metadata is accurate.

The reviewer returns one of three verdicts: *accept*, which promotes the task to the benchmark; *skip*, which excludes it; or *accept with fixes*, which returns actionable findings to the extractor. In the latter case, the extractor revises the candidate and resubmits it for review. This loop runs for at most 3 iterations, and only an *accept* verdict promotes a task. We use Opus 5 for decomposition and extraction and GPT-5.6 Sol for review. Appendix B.2 provides all agent prompts.

SWE-KERNEL-BENCH is a live benchmark that will grow as additional PRs are mined and extracted. The tasks reported here form its first release. We processed 363 of the 1,309 PRs that passed discovery. We prioritized `kernel-generate` PRs, which account for 317 candidates, because reproducing a kernel from scratch is expected to be more challenging than tuning an existing one. The remaining 46 form a grounded, best-first sample of `kernel-optimize` PRs, ordered by estimated extraction difficulty (easiest first) and confidence (highest first). We reserve the remaining reviewed PRs for future releases.

Decomposition produced 511 task candidates, of which 237 were accepted into the corpus. Of the remainder, 183 were skipped because a faithful source, gold, and workload harness could not be grounded in the PR, 79 reached the 3-iteration limit without acceptance, 5 ran out the extractor's wall-clock budget, and 7 were dropped by a deterministic audit that rechecks the commit and file blob identifiers each review reports against the repository.

To further validate extraction quality, two authors independently inspected a random sample of 20 of the 161 released tasks, using the provenance records the pipeline produces for each entry. Both authors concluded that every sampled task faithfully represented its originating PR.

Each agent operates within a bounded, multi-turn session. Decomposition averaged 11.0 turns per PR, while extraction and review averaged 33.1 and 12.0 turns per iteration, respectively. Accepted tasks required 1.9 iterations on average. Mean token usage—dominated by cached input—was 0.15M per decomposition, 1.4M per extraction iteration, and 0.38M per review. Decomposition cost an average of $0.35 per PR. Across all iterations, extraction and review cost an average of $2.70 and $0.92 per task candidate, respectively. Appendix C.3 describes the cost accounting.

3.4 SPEEDUP-GATED ADMISSION

SWE-KERNEL-BENCH evaluates whether systems can reproduce performance improvements from real-world PRs. Accordingly, the final admission gate verifies that the extracted gold implementation is both correct and faster than the source. We benchmark both implementations on every workload and require matching outputs together with a geometric-mean speedup of at least $1.0\times$. §5.2 details the measurement protocol.

Of the 237 tasks accepted during extraction, 161 pass this gate and constitute the first release of SWE-KERNEL-BENCH. Among the 76 excluded tasks, 55 did not meet the speedup threshold, 13 failed source–gold correctness checks, 5 used input generators that ignore the supplied random seed, and 3 raised errors during benchmarking.

4 SWE-KERNEL-BENCH

Following the construction process in §3.1, the first benchmark release comprises 161 tasks, each containing a source implementation, a gold implementation, and a set of workloads. Table 1 summarizes the dataset composition.

The tasks originate from 10 production libraries, with vLLM, SGLang, and FlashInfer contributing the majority. They span attention, mixture-of-experts routing, KV caching, quantization, speculative decoding, and a smaller set of non-LLM tasks, reflecting our focus on modern ML infrastructure.

The gold implementations comprise 88 CUDA kernels (including compiled CUDA-family extensions), 72 Triton kernels (Tillet et al., 2019), and 1 kernel in CuTe DSL. The source implementations are more diverse: PyTorch reference code constitutes the largest group, followed by unoptimized CUDA and Triton kernels, with a small number of C++ and SciPy implementations.

Table 1: Corpus composition by source repository, gold language, and source language. **Gold med.** denotes the median task-level gold speedup over the source (§5.2). **Source** languages are grouped as PyTorch, CUDA-family (including CuPy), Triton, or **Other** (C++ and SciPy). *Other repositories comprise CuPy, JAX, and RAPIDS cuML, each contributing fewer than 5 tasks.

		Gold			Source				
Repository	**Tasks**	**CUDA**	**Triton**	**CuTe**	**PyTorch**	**CUDA**	**Triton**	**Other**	**Gold med.**
vLLM	57	28	28	1	22	21	14	0	2.19×
SGLang	25	12	13	0	9	8	8	0	1.67×
FlashInfer	24	21	3	0	11	12	1	0	3.17×
TensorRT-LLM	14	10	4	0	10	2	0	2	2.34×
TransformerEngine	13	9	4	0	9	3	0	1	1.76×
Liger-Kernel	12	0	12	0	6	0	6	0	1.59×
pytorch/ao	10	2	8	0	3	1	6	0	3.08×
Other*	6	6	0	0	0	4	0	2	40.45×
Total	**161**	**88**	**72**	**1**	**70**	**51**	**35**	**5**	**2.19×**

The gold implementations achieve a median speedup of $2.19\times$ and a geometric-mean speedup of $3.68\times$, with a pronounced upper tail. CUDA targets offer greater median optimization headroom than Triton targets ($2.69\times$ versus $2.03\times$; Appendix A.4).

Live Benchmark As a live benchmark, SWE-KERNEL-BENCH will evolve through versioned releases published every few months. Each version will remain immutable and publicly available to preserve reproducibility. New releases will add tasks, apply deduplication, and follow a retention/retirement policy for older tasks. Results will be published in version-specific leaderboards, so every reported score is tied to a well-defined benchmark snapshot.

5 RESULTS: SWE-KERNEL-BENCH LEADERBOARD

We evaluate agents on their ability to reproduce the performance improvements achieved by the gold implementations in the first release of SWE-KERNEL-BENCH (§4). We first describe the evaluated agent configurations (§5.1) and the protocols for measuring correctness and latency (§5.2), and then present the resulting leaderboard and analysis (§5.3).

5.1 AGENTS

Each agent receives only the source (`source.py`) and must produce an optimized solution (`solution.py`); both the gold implementation and the complete workload set are withheld. Appendix B.3 provides the agent prompt.

We evaluate two configurations to isolate the value of hardware feedback. In the *oneshot* setting, the agent has no GPU access and therefore cannot execute its solution. In the *gpu-tools* setting, the agent can invoke four tools on the target GPU:

- `compile` executes the candidate to detect syntax, compilation, and kernel-launch errors.
- `check` runs the source and candidate on agent-specified inputs and reports their maximum absolute output error.
- `time` verifies correctness, measures median candidate latency, and reports speedup relative to the source.
- `NCU` profiles candidate execution with NVIDIA Nsight Compute, reporting per-kernel latency, compute and DRAM throughput, occupancy, register and shared-memory usage, roofline utilization, dominant warp-stall causes, and a derived memory- or compute-bound classification.

The *gpu-tools* setting imposes a shared budget of eight tool calls, which the agent may allocate freely across the four tools.

We evaluate Claude Opus 5, GPT-5.6 Sol, Gemini 3.1 Pro, Qwen3.5-397B-A17B-FP8, gpt-oss-120b, MiniMax-M2.5, DeepSeek-V4-Pro, and Kimi-K2.5. In terms of agent harnesses, Opus 5 uses Claude Code, GPT-5.6 uses Codex, and other models use mini-SWE-agent (Yang et al., 2024).

5.2 Benchmarking Methodology

A common benchmark driver evaluates every task by comparing a reference implementation with a candidate—for example, source against gold, or gold against an agent solution—over the task's workloads. It determines whether the candidate is correct, which implementation is faster, and the resulting speedup.

Correctness For each workload, the driver samples 10 inputs and verifies that the reference and candidate outputs agree within a predefined tolerance. A candidate proceeds to timing only if it passes every sampled input for every workload. Using multiple inputs reduces the risk of accepting implementations that are correct only for particular input values.

Performance For each workload, candidate speedup is the ratio of reference latency to candidate latency. Per-task speedup is the geometric mean across workloads. The same source-to-gold measure defines the benchmark admission criterion described in §3.4.

For each correct solution, *gold-relative speedup* is the geometric mean across workloads of gold latency divided by solution latency. We aggregate it across correctly solved tasks with another geometric mean; $1.0\times$ denotes gold parity, with lower values slower and higher values faster than gold. A solution *passes* a task if it is correct on every workload and has gold-relative speedup of at least $1.0\times$. Comparing against gold therefore measures whether the agent can reproduce an optimization that is competitive with the one shipped in the original PR.

Timing protocol Each implementation receives 10 warm-up calls per workload. The driver then performs 101 interleaved measurement iterations. In each iteration, it samples an input, overwrites 256 MB to flush the L2 cache, times the reference using CUDA events, flushes the cache again, and times the candidate on the same input. It discards the first iteration and reports the median of the remaining 100. PyTorch implementations are measured with `torch.compile` (Ansel et al., 2024) to approximate optimized production execution (Zhang et al., 2026; Yang et al., 2026), whereas CUDA and Triton implementations are benchmarked as written to avoid redundant dispatch overhead. All measurements use a single NVIDIA H100.

Warm-up excludes one-time costs such as compilation, autotuning, workspace allocation, and GPU clock ramp-up. Cache flushing prevents residual L2 state from influencing subsequent measurements. Interleaving reference and candidate measurements is important on shared GPU clusters, where non-administrative users cannot lock GPU clocks as in Lin et al. (2026); because both measurements are collected by the same job on the same device in close temporal proximity, correlated clock and thermal drift largely cancels in their ratio.

5.3 Leaderboard

Figure 4 compares agent performance on SWE-KERNEL-BENCH under the *oneshot* and *gpu-tools* configurations. Appendix C.1 reports the underlying results together with each configuration's mean cost, token usage, and number of model turns.

As seen in the figure, systems built on frontier models like Opus 5 and GPT-5.6 achieve high correctness – up to 91%. At the same time, their pass rates are low, with only 35%–54% of tasks matching or exceeding gold performance. Systems built on open-weight models perform significantly worse: DeepSeek-V4-Pro passes 10%–14% of tasks, and gpt-oss-120b matches gold on only 2%. Agent rankings remain stable across different choices of pass threshold (Appendix C.2).

Each system is evaluated once per task. Given a fixed budget, we prioritize broader coverage of systems and tasks over repeated attempts, enabling a wider cost-equivalent comparison.

GPU feedback consistently improves correctness. For example, Kimi-K2.5 rises from 41% correctness in *oneshot* mode to 70% in *gpu-tools* mode. This gain is smaller for stronger models: Gemini

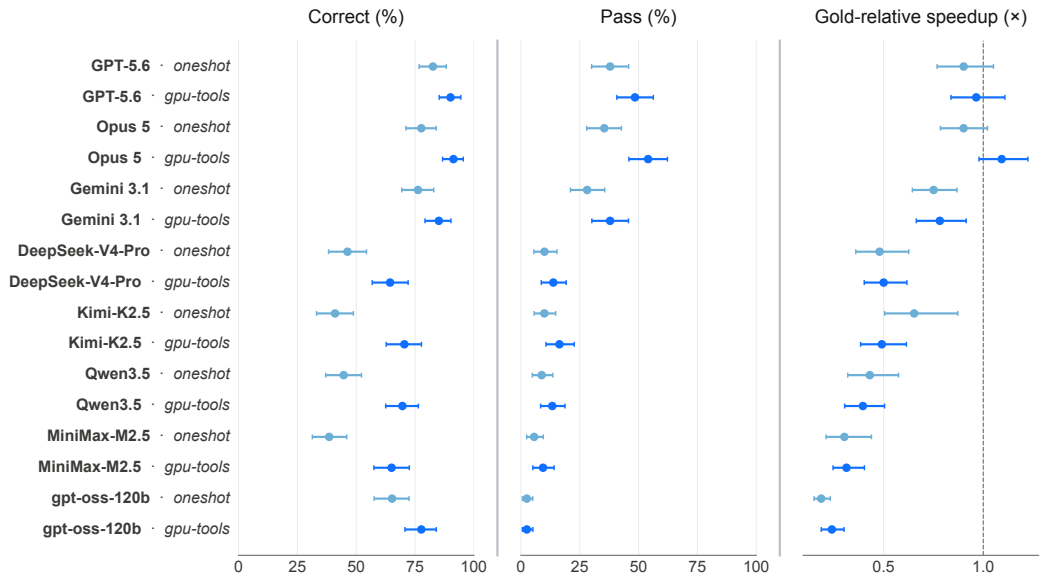


Figure 4: Kernel-optimization performance on SWE-KERNEL-BENCH by model and harness configuration (*oneshot* or *gpu-tools*; §5.1). We report the fraction of tasks solved correctly (**Correct**; left), the fraction matching or exceeding gold performance (**Pass**; center), and the geometric-mean **gold-relative speedup** among correct solutions (right). Horizontal lines show bootstrapped 95% confidence intervals. Appendix Table 6 reports detailed results and token-usage statistics.

3.1 Pro, GPT-5.6 Sol, and Opus 5 produce correct solutions in *oneshot* mode for more than 75% of tasks, despite the absence of hardware execution feedback.

Figure 4 also reports *gold-relative speedup* over correctly solved tasks. This metric follows the same overall pattern, with frontier models outperforming the evaluated open-weight models. Figure 5 shows the corresponding per-task speedup distributions.

Contamination Because every task is derived from a public merged pull request, a model could potentially recall gold code encountered during training. We assign each task its pull request merge date, divide the tasks into four equal-sized temporal bins, and compare each agent's pass rates before and after its reported training cutoff. When no cutoff is disclosed, we use the model's earliest public release as a conservative proxy (Figure 12, Appendix C.4). Memorization would predict higher pass rates on earlier, potentially observed tasks. Instead, pass rates remain flat or increase on later tasks for every model and harness configuration, providing no evidence that contamination drives the leaderboard results.

Correlates of success The source-to-gold speedup is weakly associated with agent correctness (Spearman $\rho = -0.008$) and pass rate ($\rho = -0.171$), but moderately negatively associated with agent gold-relative speedup ($\rho = -0.468$). Thus, tasks with greater source-to-gold improvement leave agents less likely to approach or exceed gold performance. We also observe moderate inter-agent agreement in per-task gold-relative speedups (Appendix C.5).

The source language further affects performance (Appendix C.6): tasks with PyTorch sources generally have lower pass rates. There are also some system-specific patterns; for example, DeepSeek-V4-Pro performs slightly better on Triton-source tasks.

Unsolved tasks In 32 of the 161 tasks (20%), none of the evaluated systems successfully match gold; for 30 of these tasks, at least one system produces a correct solution, indicating that the primary challenge is producing an optimized implementation. This subset of tasks is characterized by greater source-to-gold headroom than the full corpus (median $3.01\times$ versus $2.19\times$). The release explicitly tags these tasks as a standing target for tracking future progress.

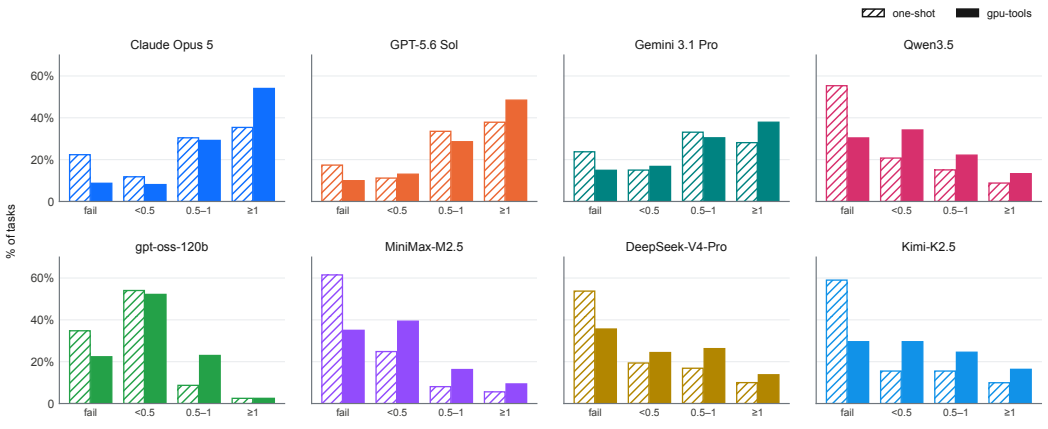


Figure 5: Per-task distributions of gold-relative speedup – gold latency divided by solution latency.

Tool use In *gpu-tools* mode, all agents devote a large fraction of their tool budget to `check`, which evaluates correctness on agent-provided probes (Appendix C.7). Stronger agents allocate more timing calls, whereas weaker ones rely more heavily on basic execution and compilation checks.

6 Discussion

The first release of SWE-KERNEL-BENCH demonstrates that current coding agents still struggle to match production kernel optimizations. Even the strongest configuration produces correct outputs on 91% of attempted tasks but reaches the pass threshold of $1.0\times$ gold-relative speedup on only 54%. This gap may narrow as agents improve, while production optimization practice will also evolve. Periodically mining new PRs keeps the benchmark aligned with current production work.

Approaches to benchmark construction trade controlled coverage for grounding in observed practice. Top-down suites such as KernelBench (Ouyang et al., 2025) prescribe operators, enabling systematic coverage and targeted capability tests. In contrast, SWE-KERNEL-BENCH mines merged PRs, targeting optimizations that domain experts considered useful for production. Automated mining scales this bottom-up approach across repositories, capturing real-world optimization work without restricting it to a predefined task inventory.

Kernel optimization is particularly amenable to bottom-up construction. The tasks that SWE-KERNEL-BENCH extracts expose compact numerical interfaces, admit synthetic tensor inputs, and support common correctness and latency measurements across implementation languages. These properties allow a shared driver to evaluate heterogeneous optimizations at scale. Beyond kernels, this approach may extend to domains with isolatable changes and executable correctness and performance criteria; other domains would require more task-specific infrastructure.

7 Conclusion

SWE-KERNEL-BENCH is a live kernel generation benchmark built from merged performance PRs. A fully automated pipeline discovers candidate changes, extracts each optimization into a standalone task with its pre-optimization source, maintainer-accepted implementation, and workloads, and validates the task on the target hardware, requiring neither a prescribed PR format nor task-specific manual engineering. Its first release contains 161 CUDA and Triton tasks from 10 production libraries, each evaluated by how closely a correct agent solution approaches the shipped implementation's performance. Evaluating eight contemporary agentic systems with this measure reveals substantial headroom between the performance of correct agent solutions and the production gold implementations. By grounding evaluation in production optimization work and incorporating new PR contributions as they land, SWE-KERNEL-BENCH provides a renewable measure of whether coding agents can keep pace with the evolving frontier of GPU kernel performance.

REPRODUCIBILITY STATEMENT

To support reproducibility, we provide in the supplementary material the full code and associated resources required to reproduce both the benchmark construction process and the results reported in this paper. This includes the Python code for benchmark generation and evaluation, together with the agent prompts used for discovery, mining, decomposition, extraction, review, and solution generation. The benchmark itself is also included in the supplementary material; for each task, it contains the unoptimized source implementation, the corresponding optimized gold implementation, the evaluation workloads, and metadata describing the upstream pull request from which the task was extracted. We additionally provide the solutions generated by the eight agentic systems evaluated on the benchmark, the measured speedup of each solution, and a leaderboard summarizing the benchmark results. The full project will additionally be released as an open-source library, and, as a live benchmark, will be continuously maintained and updated with newly added tasks and associated materials made publicly available. The released artifact is described in Appendix D.

AI USE STATEMENT

The released corpus is model-generated: models categorize mined pull requests, gate candidates, decompose them into variants, and extract and review each task's source, gold, and workload harness files (§3.2, §3.3). Every released task is therefore model-written code derived from a merged pull request.

We used generative AI to implement methods and write code: the construction pipeline, the scoring driver, and the analysis scripts that produce every number, table, and figure in this paper. We also used it to draft and copy-edit parts of this paper and to search and summarize literature, which we then verified against the cited works. We did not use it to propose or refine hypotheses, to design or critique the methodology, or to interpret results.

We have reviewed all AI-assisted work: no reported quantity is model-written, since every live number is a macro emitted by an analysis script reading the corpus. The construction and scoring code was read and exercised by the authors, and AI-drafted prose was revised by an author and checked against its cited source. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024. doi: 10.1145/3620665.3640366.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=nMpJoVmRy1.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA agent: Large-scale agentic RL for high-performance CUDA kernel generation, 2026. URL https://arxiv.org/abs/2602.24286.

Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. From large to small: Transferring CUDA optimization expertise via reasoning graph. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=vqESUhcSOG.

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 *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=pNxw4i5Dp2.

Chenxi Huang, Alex Mathai, Feiyang Yu, Aleksandr Nogikh, Petros Maniatis, Franjo Ivančić, Eugene Wu, Kostis Kaffes, Junfeng Yang, and Baishakhi Ray. Outrunning LLM cutoffs: A live kernel crash resolution benchmark for all. In *Forty-third International Conference on Machine Learning*, 2026a. URL https://icml.cc/virtual/2026/poster/63308.

Jinjun Huang, Zhongzhen Wen, Tongtong Xu, Meng Yan, Xin Xia, and Zhongxin Liu. RealisticTritonBench: A benchmark for Triton-kernel generation in real-world AI frameworks. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2026b. doi: 10.1145/3832783.3837457. URL https://arxiv.org/abs/2608.12004. To appear.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

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

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

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, 2026. URL https://arxiv.org/abs/2603.19173.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. SWE-fficiency: Can language models optimize real-world repositories on real workloads? In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0pyFbZSfbT.

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*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-agents. In *The Thirty-ninth Annual*

Conference on Neural Information Processing Systems Datasets and Benchmarks Track, 2025. URL https://openreview.net/forum?id=I5qDL315bQ.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973.

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 of *Proceedings of Machine Learning and Systems*, pp. 2016–2064. MLSys, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/file/37e44c4b5321605735be9761f9b758fc-Paper-Conference.pdf.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

Lingyun Yang, Yuxiao Wang, Shenghao Liang, Linfeng Yang, Daocheng Ying, Chunbo You, Rui Zhang, Luping Wang, Yinghao Yu, Guodong Yang, and Liping Zhang. Are LLM-generated GPU kernels production-ready? a trace-driven benchmark and optimization agent, 2026. URL https://arxiv.org/abs/2607.14541.

Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, Wentao Zhang, Chunlei Men, Guang Liu, and Yonghua Lin. Towards automated kernel generation in the era of LLMs. In Diego Calvanese (ed.), *Proceedings of the Thirty-Fifth International Joint Conference on Artificial Intelligence, IJCAI-26*, pp. 8069–8077. International Joint Conferences on Artificial Intelligence Organization, 8 2026. doi: 10.24963/ijcai.2026/895. URL https://doi.org/10.24963/ijcai.2026/895. Survey Track.

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 *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=OGWkr7gXka.

Yunxiang Zhang, Ping Yu, Jianyu Wang, Max Xiangjun Fan, Julian Reed, Azalia Mirhoseini, and Will Su. KernelBench-Verified: Do LLM-generated kernels actually beat PyTorch?, 2026. URL https://arxiv.org/abs/2607.16241.

A THE BENCHMARK DATASET

This appendix collects the corpus-level detail behind the benchmark: the per-workload measurements of the example task of Figure 2, the repositories mined to build it, the category taxonomy the discovery stage assigns over, and the gold-speedup headroom of the admitted corpus.

A.1 EXAMPLE TASK

Table 2 reports the per-workload measurements of the example task whose source and gold code is shown in Figure 2.

Table 2: Per-workload performance for the task of Figure 2; all workloads pass correctness checks. Latencies are in milliseconds on an NVIDIA H100.

Workload	Correct	Compiled (ms)	Gold (ms)	Speedup
`b1_l1_hq8_hk8_off0_d128`	yes	0.149	0.028	5.25×
`b19_l19_hq8_hk8_off15_d64`	yes	0.155	0.037	4.14×
`b99_l19_hq16_hk8_off99_d256`	yes	0.168	0.039	4.28×
`b99_l204_hq16_hk8_off0_d128`	yes	0.215	0.105	2.06×
`b989_l4_hq16_hk8_off99_d128`	yes	0.174	0.049	3.53×
`b989_l204_hq16_hk8_off0_d128_llama31`	yes	0.967	0.860	1.12×

A.2 MINED REPOSITORIES

Table 3 lists all 23 repositories passed as input to the task discovery phase (see §3.2). That phase mined merged PRs from January 2023 onwards. A few high-volume repositories were capped to their most recent 2,000 PRs. Some repositories eventually contributed no tasks to the released benchmark.

A.3 CATEGORY TAXONOMY

The open-weights categorizer of §3.2 assigns each filtered PR one or more of the categories in Table 4; its prompt is reproduced in Appendix B.1.

The taxonomy separates kernel authorship from adjacent forms of GPU engineering. The first five categories concern executable kernels. Memory, compiler, and dispatch cover surrounding runtime decisions; build and quantization may not produce an isolatable kernel; and distributed changes often require a multi-device harness. Fusion and existing-kernel optimization fit the current standalone before/after contract most directly. Compiler, build, and distributed changes would require broader task and measurement interfaces.

A.4 HEADROOM

Table 5 gives the distribution of task-level gold speedup over the admitted corpus, split by target language. The median task is 2.19× faster than its source, a long upper tail pulls the geometric mean to 3.68×.

B PROMPTS

This appendix reproduces the agent prompts used across the pipeline, grouped by stage: discovery (§3.2), extraction (§3.3), and evaluation (§5). Each figure is abridged; the full prompt text is released verbatim in the supplementary material.

B.1 DISCOVERY

Discovery uses two prompts in sequence. The open-weights categorizer (Figure 6) assigns each filtered PR one or more of the categories in Table 4. The frontier-review gate (Figure 7), run with

Table 3: Discovery seed repositories. For each repository, we report the number of PRs scanned, the merge-date range from the earliest to latest scanned PR, and the number of benchmark tasks contributed. Rows are ordered first by tasks contributed and then by PRs scanned.

Repository	Role	PRs scanned	First	Last	Tasks
vllm	serving	18,001	2023-03-01	2026-07-07	57
sglang	serving	6,700	2024-01-09	2026-07-07	25
flashinfer	kernel library	2,000	2023-11-11	2026-06-17	24
TensorRT-LLM	serving	7,301	2023-10-18	2026-07-07	14
TransformerEngine	training library	1,850	2023-01-03	2026-06-17	13
Liger-Kernel	kernel library	746	2024-08-09	2026-06-16	12
ao	quantization	2,674	2023-11-10	2026-07-07	10
cupy	array library	1,629	2023-01-04	2026-06-17	4
jax	array/compiler	2,000	2024-11-13	2026-06-17	1
cuml	ML library	1,753	2023-01-02	2026-06-16	1
triton	compiler/DSL	2,000	2024-10-24	2026-06-17	0
cudf	dataframe library	2,000	2024-01-12	2026-06-17	0
composable_kernel	kernel library	2,000	2023-07-15	2026-02-03	0
xgboost	ML library	1,946	2023-01-02	2026-06-16	0
pytorch	framework	1,261	2023-01-04	2026-06-16	0
cutlass	kernel library	524	2023-01-04	2026-06-16	0
flash-attention	kernel library	450	2023-01-07	2026-06-17	0
xformers	kernel library	111	2023-01-05	2026-06-15	0
mamba	model library	89	2023-12-04	2026-06-15	0
DeepGEMM	kernel library	83	2025-02-26	2026-06-04	0
ThunderKittens	kernel library	67	2024-03-13	2026-05-27	0
heavydb	database	20	2023-01-02	2025-06-17	0
FBGEMM	kernel library	5	2025-03-20	2025-10-21	0
Total		**55,210**			**161**

Table 4: Categories used by the categorization prompt: 11 kernel-change categories and one rejection label.

Category	Description
`kernel_generate`	Write a new GPU kernel, including its algorithm, launch, and memory access.
`kernel_fuse`	Combine existing operations or kernels to remove launches and intermediates.
`kernel_split`	Decompose a large kernel into cooperating stages, such as split-K or multi-pass work.
`kernel_remove`	Eliminate a custom kernel through redundancy removal or a library replacement.
`kernel_optimize`	Tune an existing kernel without changing its logical computation.
`kernel_memory`	Change allocation, buffering, streams, graphs, copies, or synchronization.
`kernel_compiler`	Modify GPU compiler IR, lowering, code generation, or JIT infrastructure.
`kernel_dispatch`	Change backend selection, capability gates, fallbacks, or device placement.
`kernel_build`	Change GPU build, test, benchmark, CI, logging, or profiling infrastructure.
`kernel_quant`	Change quantization, scaling, calibration, mixed precision, or precision dispatch.
`kernel_distributed`	Change collectives, sharding, parallelism, or cross-device synchronization.
`reject_llm`	The PR is not about GPU kernels or GPU optimization.

Opus 5, then decides benchmark candidacy. Two properties of the gate are visible in the prompt but not in the body. The chosen category and the benchmark-candidacy flag come from a single decision: a PR judged unsuitable is assigned an empty category, so labelling and admission are one step rather than two. And target hardware is explicitly declared out of scope, which is what keeps kernels for Hopper, Blackwell, and vendor accelerators in the corpus rather than screening them out before extraction fixes the target architecture.

You classify GitHub pull requests into GPU optimization categories.
Given a PR title, description, detected signals, and (optionally) a prior heuristic classification to verify, assign one or more categories:
[the 11 kernel-change categories and `reject_llm`, each with a one-to-two sentence definition; see Table 4]
Also identify GPU backends: `cuda`, `triton`, `hip`, `jax`, `sycl`, or others.
Provide a 1–2 sentence summary of what the PR does from a GPU/performance optimization perspective.
Respond ONLY with valid JSON matching this schema:
`{"categories": ["..."], "gpu_backends": ["..."], "confidence": 0.0-1.0, "reasoning": "one sentence", "summary": "...", "new_suggested_category": ""}`
The `new_suggested_category` field should hold a suggested `snake_case` name when the PR fits none of the existing categories, and is empty otherwise.

Figure 6: The prompt used by the open-weights categorizer (§3.2). The category definitions the model selects from are elided here and listed in Table 4. The model returns one or more categories per PR. The figure is abridged; the full prompt is released verbatim in the supplementary material.

You are a second-pass reviewer for GPU kernel PRs. Your job is to decide whether a PR is a good candidate for a kernel benchmarking suite (SWE-kernel-bench) AND, if so, which single benchmarkable category it best fits.
The benchmarkable categories you may choose from are `kernel_generate` and `kernel_optimize`. Pick the ONE category that best describes the core change. If the PR fits neither, or is not a good benchmark candidate, set `category` to `""` and `bench_candidate` to false.
A good candidate has ALL of these properties:
(1) **Clear before/after**: an identifiable BEFORE implementation that the PR replaces with a faster AFTER — a naive Python/library path (for a newly written kernel), or an existing, already-real GPU kernel at the pre-PR revision (for an optimization or fusion of existing kernels).
(2) **Self-contained computation**: an isolated, extractable computation, not deeply entangled with framework state, model weights, or a full inference pipeline.
(3) **Benchmarkable**: exercisable with synthetic inputs (random tensors, generated data); it does NOT require real model weights, trained parameters, or a running server.
(4) **Meaningful speedup**: the PR demonstrates or claims a measurable performance improvement of the AFTER over the BEFORE.
The **target hardware is NOT a suitability criterion**: kernels for any GPU architecture are eligible, including Ampere (SM80), Hopper (SM90), Blackwell (SM100), and vendor-specific accelerators (NPU, TPU, AMD GFX). Do NOT reject a PR because of the hardware it targets; the target architecture is determined later, during extraction.
Respond with JSON: a `category` (one allowed name, or `""`); a `bench_candidate` boolean, true only if all four criteria hold and independent of target hardware; a one-sentence `bench_candidate_reason`; a `source_type` (`pytorch` when the before path is a naive framework/library implementation, else `triton`/`cuda` when it is already a real kernel of that language); a `target_type` (`triton` or `cuda`); a `baseline_description` and `kernel_description`; an `extraction_difficulty` (`easy`/`medium`/`hard`); and a `confidence` in $[0, 1]$.

Figure 7: The frontier-review gate prompt (§3.2), run with Opus 5. The allowed-category placeholder is resolved to the two pursued categories, and the per-field response guidance is summarized. The figure is abridged; the full prompt is released verbatim in the supplementary material.

Table 5: Distribution of task-level gold speedup (×) over the admitted corpus. Each task is first aggregated geometrically over workloads. The per-language rows cover 160 of the 161 tasks in the All row; CuTe (1 task) is omitted as too small a sample for a distribution.

Target	n	P10	P25	Median	P75	P90	Geomean
All	161	1.08	1.27	2.19	5.48	35.46	3.68
CUDA	88	1.08	1.26	2.69	8.21	57.38	4.38
Triton	72	1.09	1.40	2.03	4.04	12.80	2.83

B.2 EXTRACTION

The task-extraction pipeline of §3.3 is three prompted agents, run in sequence on each PR. The decomposition prompt (Figure 8) triages a PR into one or more independently benchmarkable optimizations, emitting a `focus` for each and writing no benchmark code. The extraction prompt (Figure 9) turns one such focus into the runnable source, gold, and workload harness files under a shared call signature; its central constraint is the grounded-source invariant, which forbids the fabricated baseline that is the most common way an extraction inflates a speedup. The review prompt (Figure 10) audits the extracted entry against the repository and returns the structured verdict that either promotes the task or feeds findings back into another extraction pass. The extractor and reviewer are drawn from different model families, so the audit is not the extractor grading its own work.

You are triaging a GitHub PR before it is turned into GPU-kernel benchmark entries. Your ONLY job is to decide **how many independently-benchmarkable kernel optimizations the PR contains**, and to describe each one. You do NOT write any benchmark code — a later step extracts each optimization. Read the PR at `$pr_url` with `gh`/`git`: its title, description, and the actual diff.
What counts as a separate variant. Split the PR into separate variants ONLY when it contains distinct optimizations that can each be benchmarked on their own — a different kernel, a different algorithm or code path, or a different operation, each with its own before/after and its own inputs. Two kernels added in one PR (a CSR path and a COO path), the same operation exposed via two independently selectable strategies, or two genuinely different operations ARE multiple variants. One kernel plus its launcher/autotune/config, refactors or plumbing or tests around a single kernel, and micro-variations of tile size or dtype (those are workloads) are NOT. **Default to a single variant**: most PRs are one optimization, and you should return more than one only when confident each is independently benchmarkable and materially distinct.
Output. Write two files and nothing else. `decomposition.json` is a JSON array with one object per variant, most important first, each carrying a `variant_key` (a short, stable, lowercase slug, unique in the array and derived from the code rather than from ordering); a `focus` object with three required fields — `target` (the AFTER kernel or function and the file it lives in), `source_baseline` (what the BEFORE is and where it comes from: the pre-PR path it replaces or the in-repo reference it is checked against, said explicitly when the baseline is an existing hand-written kernel rather than a high-level reference), and `exclusions` (every sibling variant plus anything rejected as not separately benchmarkable, one short phrase each naming the function or file); and optional `target_function`, `source_type`, and `target_type`. An ordinary one-optimization PR still returns a single-element array with a full `focus`, since naming its target and baseline is what keeps the extraction from drifting; a variant whose `focus` is missing a field, or is a plain string rather than an object, is discarded and the PR left undecomposed.
`decomposition.md` records, in a dozen lines, what the PR changes, why this many variants, and what was left out, so a reviewer can judge the split later without re-reading the trace.

Figure 8: The PR-decomposition prompt (§3.3), the first agent of the extraction pipeline. It emits one `focus` per separately-benchmarkable optimization and writes no benchmark code; the field-by-field JSON guidance and the worked example are summarized here. The figure is abridged; the full prompt is released verbatim in the supplementary material.

Convert a GitHub PR that changes a GPU kernel into a self-contained benchmark entry: a `source_impl` (BEFORE) versus an `optimized_impl` (AFTER) that the driver times on identical inputs. **Faithfulness to the PR is everything.** You are given the PR at `$pr_url` and, when the PR was decomposed, a Focus naming the one optimization to extract and what to exclude.
Grounded source (the central invariant). `source.py` must be real, runnable code from exactly one of three origins: the pre-PR implementation at the parent commit; a reference implementation the PR itself adds; or a named standard-library/CPU routine the PR explicitly measures against and that the code actually calls. Never fabricate a strawman baseline and never downgrade an already-fast kernel into a naive loop — an unfair source manufactures the speedup. `target.py` must contain the real kernel the PR introduces or improves (Triton/CUDA/...), not a library wrapper or a pure-`torch` reimplementation.
Procedure. (1) Analyze the PR with `gh`. (2) Clone and read the full changed files at BOTH the base and PR refs — hunks alone are insufficient. (3) Emit `source.py`, `target.py`, and `infra.py` (a `Workload` dataclass, `FULL_WORKLOADS` and `QUICK_WORKLOADS`, and `generate_inputs`) under one shared, keyword-only call signature. (4) Self-verify on real hardware: a GPU tool compiles both sides and checks numerical equivalence over the workloads; **timing is withheld** so the definition cannot be tuned toward a favorable result. If the two sides disagree, the only honest resolutions are to fix an ungrounded source, move the inputs toward the PR's own, or — when both are already grounded and still disagree — skip: never loosen tolerances, drop failing workloads, or narrow the returned outputs so the disagreement stops being compared.
Workloads must trace to the PR's own benchmarks, tests, or changed code — shapes, dtypes, and the input *construction* (distribution, range, legality), never invented, and never seeded inside `generate_inputs`. **Timed-region parity:** neither side may hide or add work (no host transfers, caching, or double-compilation), the sole exception being the inherent transfer of a CPU-library baseline. **No installs and no network** anywhere in the entry. Also emit `extraction_sources.yaml` (every `(repo_path, ref, lines)` each file drew from, off a numbered view of the cited revision) and `metadata_output.yaml` (`source_type`, `target_type`, and the accurate *minimum* `architecture`). If the PR is not faithfully extractable — no grounded source, not a kernel-level optimization, not liftable from framework internals, or not equivalent to itself — write `EXTRACTION_SKIP.json` with a reason and a category rather than a fabricated entry.

Figure 9: The extraction prompt (§3.3), run with the extractor model. The grounded-source invariant, the disagreement-resolution rules, the harness contract, and the available-dependency list are summarized; the code conventions and worked examples are omitted. The figure is abridged; the full prompt is released verbatim in the supplementary material.

You are an expert GPU-kernel benchmark auditor. You are given an entry another agent extracted from a GitHub PR — `source.py`, `target.py`, `infra.py`, and `metadata.yaml` — plus the source PR. Judge, **skeptically and with evidence from the real repository**, whether the entry faithfully represents the PR and constitutes a valid, fair speedup benchmark, or whether the PR does not fit extraction and should be skipped. You are a reviewer, not an author: do NOT rewrite the entry. **The PR and the repository are your ground truth — not `metadata.yaml`**, every field of which is an unverified claim by the extractor. From the diff, first decide which archetype the change is (optimizes an existing kernel; introduces a kernel replacing a pre-PR baseline; introduces a kernel with its own reference; or replaces a standard-library/CPU baseline), since that defines what a faithful source looks like. Then, **mandatory**: clone the repository, fetch the PR head, resolve the base commit, and read the *full* changed files at both refs — judging from patch hunks alone is not acceptable. Record the base commit and the before/after git blob ids of each file you read; these are checked against the repository, so source and target faithfulness may be rated `pass` only if you actually cloned and read both sides.
Rubric (rate each PASS/CONCERN/FAIL with evidence, treating "cannot verify" as CONCERN): *source faithfulness* — a grounded source exists and matches its origin, with no strawman and no leakage of the optimization; *target faithfulness* — the real kernel of the PR, all its relevant improvements and no unrelated ones, not a library wrapper; *fairness and equivalence* — identical signatures, both sides return everything the timed region computes, timed-region parity, and no installs or network; *workload faithfulness* — shapes, dtypes and inputs grounded in the PR, exercising the optimized path, with fresh per-call randomness; and *metadata and classification* — accurate types and accurate minimum architecture, and the PR is genuinely a kernel-level optimization.
Output a short prose assessment followed by a single JSON block: a `verdict` of `accept`, `accept_with_fixes`, or `skip`; a `confidence`; the `ground_truth` block; a status for each of the five `dimensions`; a severity-ranked `findings` list (each `blocker`/`major`/`minor`, with dimension, issue, evidence, and a suggested fix); and, on a skip, a `skip_reason` and a `skip_category`. `accept` only if the entry is clean as-is; `accept_with_fixes` if it is salvageable but has at least one actionable finding and no blocker; `skip` if any skip criterion holds. Do not invent findings to appear thorough, and do not pass an entry you could not verify against the repository.

Figure 10: The extraction-review prompt (§3.3), run with a reviewer from a different model family than the extractor. The per-item rubric guidance and the JSON schema detail are summarized; the mandatory clone-and-read procedure and the accept/skip rules are shown. The figure is abridged; the full prompt is released verbatim in the supplementary material.

B.3 EVALUATION

Figure 11 gives the prompt every leaderboard system of §5 is given to produce a candidate solution. It fixes the leaderboard's information boundary: the solver sees only the source and the task's call signature, writes `solution.py`, and never sees the gold or the full workloads it will be scored on. The correctness contract in the prompt is the same one the driver enforces at measurement time, so a solution that passes the prompt's stated bar is exactly a solution the harness will admit for timing. The hardware description and the tool-access block are placeholders resolved per entry, which is how one prompt spans both the one-shot and the iterative compile/test/profile tracks of §5.

> You have a file called `source.py` in the current directory. It contains a Python function `source_impl` that performs a computation on GPU tensors using PyTorch.
> **Autonomy.** You are running fully autonomously in a headless session, with no human to answer questions or approve a plan. Do not enter plan mode and do not ask anything; write `solution.py` directly.
> **Task.** Read `source.py` to understand its computation, then write an optimized `solution.py` that computes the same result faster. It must define `solution_impl` with the exact same signature as `source_impl` and return numerically equivalent results within tolerance.
> You have only `source.py` — you do not know the specific input shapes the benchmark will run, so write an implementation that is correct and fast across the general shape range implied by the code (arbitrary sizes, masked boundaries, no hard-coded dimensions). Optimize however works best on this hardware: a Triton kernel, CUDA C++ compiled in-file via `load_inline`, a faster or better-fused library call, a smarter algorithm, or any combination.
> *[The hardware description and the set of self-verification tools available to the solver are supplied here, and vary per leaderboard entry.]*
> **Requirements.** Match `source_impl`'s signature, including keyword-only arguments; a function that mutates in place must return the mutated tensor; a function that returns a tuple must match the non-`None` element count; preserve output shapes exactly. No hidden fallback paths: the returned implementation is the one that runs and is timed — no `try/except` that silently substitutes a slower path, no capability guards that swap in a different implementation. Do not fake correctness by hard-coding, caching, or aliasing the reference buffer.
> **Correctness bar.** Correctness is checked against `source_impl` with `torch.allclose` over multiple reseeded trials at the driver tolerance (`atol=rtol=`10^{-2} for fp16/bf16, 10^{-4} for fp32). Accumulate reductions in fp32 even when inputs are lower precision, then cast back, and match the numerics `source_impl` actually uses.
> Do not look for the reference solution (`target.py`) or the benchmark's actual workloads and inputs — they are hidden on purpose. Base your implementation only on `source.py`.

Figure 11: The baseline solution-generation prompt (§5), given to every leaderboard system. The solver receives only the source and must write `solution.py`; the gold and the full workloads are withheld. The hardware description and tool-access blocks are placeholders resolved per entry, and the verbose optimization guidance is summarized. The figure is abridged; the full prompt is released verbatim in the supplementary material.

C EXTENDED LEADERBOARD RESULTS

This appendix expands the leaderboard of §5: the detailed per-configuration table and the cost rates behind its dollar column, a training-data contamination check, followed by cross-system speedup correlations, a breakdown by source implementation family, and the gpu-tools tool-use analysis.

C.1 DETAILED LEADERBOARD RESULTS

Table 6 reports the detailed values underlying Figure 4, together with each configuration's mean cost, token usage, and model turns.

Table 6: Results per **Model** and interaction **Mode** (oneshot or gpu-tools, see §5.1). System performance is reported with the percentage of tasks solved correctly (**%Cor**) and the percentage solved correctly with gold-relative speedup of at least $1.0\times$ (**%Pass**). Also shown is the geometric-mean **gold-relative speedup** on correctly solved tasks, where $1.0\times$ denotes gold parity, and the mean cost (**$**), tokens (**Tok.**) per task, and session **Turns**. Dollar costs are estimates for some models (see Appendix C.3).

Model	Mode	%Cor	%Pass	Gold-relative speedup	$	Tok.	Turns
GPT-5.6 Sol	oneshot	83%	38%	0.90	$0.19	54k	4.5
	gpu-tools	90%	48%	0.96	$0.42	360k	12.5
Claude Opus 5	oneshot	78%	35%	0.90	$0.69	301k	13.7
	gpu-tools	91%	54%	1.09	$0.67	287k	15.4
Gemini 3.1 Pro	oneshot	76%	28%	0.75	$0.53	230k	11.1
	gpu-tools	85%	38%	0.78	$0.46	197k	14.3
DeepSeek-V4-Pro	oneshot	46%	10%	0.48	$0.38	358k	14.4
	gpu-tools	64%	14%	0.50	$0.45	428k	24.5
Kimi-K2.5	oneshot	41%	10%	0.65	$0.05	88k	7.3
	gpu-tools	70%	16%	0.49	$0.07	194k	17.1
Qwen3.5	oneshot	45%	9%	0.43	$0.08	128k	8.8
	gpu-tools	70%	13%	0.40	$0.20	365k	24.6
MiniMax-M2.5	oneshot	39%	6%	0.30	$0.06	148k	11.5
	gpu-tools	65%	9%	0.31	$0.12	369k	27.1
gpt-oss-120b	oneshot	65%	2%	0.19	$0.01	56k	8.8
	gpu-tools	78%	2%	0.24	$0.04	216k	21.0

C.2 Sensitivity to the pass threshold

The main leaderboard counts a correct solution as passing when its gold-relative speedup is at least $1.0\times$. To test whether the conclusions depend on this precise cutoff, Table 7 re-scores the same H100 runs at thresholds from $0.90\times$ to $1.10\times$, without changing correctness or corpus admission. Absolute pass rates vary with the bar, as expected, but the leading ordering is stable: Claude Opus 5 in *gpu-tools* mode ranks first and GPT-5.6 Sol in *gpu-tools* mode ranks second at every threshold. The broader separation between *gpu-tools* and *oneshot* also remains visible across the sweep.

C.3 Model cost rates

The dollar costs in Table 6 mix each run's own reported spend, where its backend provides one, with list-price estimates for the runs whose backend reports only tokens. Where an agent backend reports its own spend for a run, that figure is used directly, since it reflects the run's real cached-input discount; otherwise the run's input and output token counts are priced at the model's public list rate, an upper bound that charges every input token at the fresh-input rate. Table 8 gives those per-model rates, with open-weights models priced at a representative public host that serves the same model version.

The extraction and review costs in §3.3 use the same rates: GPT-5.6 Sol's review cost is estimated from its reported token counts at the public list rate in Table 8.

C.4 Training-data contamination

SWE-KERNEL-BENCH is built from public merged pull requests, so a model could in principle have seen a task's gold solution during pretraining and recall it rather than optimize from first principles. We check whether this inflates the leaderboard by testing for a before/after-cutoff performance gap, the pre/post-cutoff protocol used by live code benchmarks such as LiveCodeBench (Jain et al., 2025).

Table 7: Sensitivity of leaderboard rank and pass rate to the gold-relative speedup threshold. Each cell gives rank followed by pass rate in parentheses; rates tied at the displayed precision share a rank. A pass additionally requires correctness. The ordering of the leading systems is stable: Claude Opus 5 in *gpu-tools* mode ranks first and GPT-5.6 Sol in *gpu-tools* mode ranks second at every threshold.

Model	**Mode**	**0.90**	**0.95**	**1.00**	**1.05**	**1.10**
Claude Opus 5	gpu-tools	1 (64.6%)	1 (61.5%)	1 (54.0%)	1 (44.1%)	1 (37.3%)
GPT-5.6 Sol	gpu-tools	2 (59.6%)	2 (55.3%)	2 (48.4%)	2 (40.4%)	2 (35.4%)
GPT-5.6 Sol	oneshot	3 (48.4%)	3 (44.7%)	3= (37.9%)	3 (32.9%)	3 (26.7%)
Gemini 3.1 Pro	gpu-tools	4 (47.8%)	4= (44.1%)	3= (37.9%)	4= (29.2%)	4= (24.2%)
Claude Opus 5	oneshot	5 (47.2%)	4= (44.1%)	5 (35.4%)	4= (29.2%)	4= (24.2%)
Gemini 3.1 Pro	oneshot	6 (40.6%)	6 (36.2%)	6 (28.1%)	6 (24.4%)	6 (18.1%)
Kimi-K2.5	gpu-tools	7 (24.5%)	7 (19.5%)	7 (16.4%)	7 (12.6%)	7 (11.9%)
DeepSeek-V4-Pro	gpu-tools	8 (24.4%)	8 (19.4%)	8 (13.8%)	9 (11.2%)	9 (8.8%)
Qwen3.5	gpu-tools	9 (19.6%)	9 (16.5%)	9 (13.3%)	8 (12.0%)	8 (10.1%)
DeepSeek-V4-Pro	oneshot	10 (17.5%)	10 (15.6%)	10 (10.0%)	13 (5.6%)	13 (5.6%)
Kimi-K2.5	oneshot	11 (14.3%)	11 (12.4%)	11 (9.9%)	11 (7.5%)	10= (6.2%)
MiniMax-M2.5	gpu-tools	13 (12.5%)	12 (11.2%)	12 (9.4%)	10 (8.1%)	10= (6.2%)
Qwen3.5	oneshot	12 (12.6%)	13 (10.7%)	13 (8.8%)	12 (6.3%)	12 (5.7%)
MiniMax-M2.5	oneshot	14= (7.5%)	14 (6.8%)	14 (5.6%)	14 (4.3%)	14 (3.7%)
gpt-oss-120b	gpu-tools	14= (7.5%)	15 (5.6%)	15= (2.5%)	15= (2.5%)	15= (2.5%)
gpt-oss-120b	oneshot	16 (3.7%)	16 (3.7%)	15= (2.5%)	15= (2.5%)	15= (2.5%)

Table 8: Public-host list rates (USD per million tokens) behind the **$** column of Table 6: **Input**, **Output**, and **Cached** input. Where a cached-input rate is not published separately it repeats the input rate.

Model	**Input**	**Output**	**Cached**
Claude Opus 5	self-reported by its CLI		
GPT-5.6 Sol	$5.00	$30.00	$0.50
Gemini 3.1 Pro	$2.00	$12.00	$0.20
Qwen3.5	$0.45	$3.00	$0.22
gpt-oss-120b	$0.15	$0.60	$0.15
MiniMax-M2.5	$0.30	$1.20	$0.03
DeepSeek-V4-Pro	$1.74	$3.48	$0.14
Kimi-K2.5	$0.60	$3.00	$0.10

We date each task by the merge date of its source pull request—the point at which the gold optimization became public—and sort the published tasks into four equal-count bins. Because the tasks are concentrated in recent months, we plot bins against task-date rank rather than calendar time (Figure 12), so equal-count bins are equally spaced and each model's boundary lands at its rank among the tasks regardless of the calendar skew. For each system (model $\times$ harness configuration) we compute the pass rate—the fraction of tasks solved correctly with gold-relative speedup of at least $1.0\times$—within each bin.

The boundary for each model is its published training-data cutoff where one exists: Opus 5, GPT-5.6, Gemini 3.1, and gpt-oss-120b (solid line). The remaining four models—DeepSeek-V4-Pro, Kimi-K2.5, Qwen3.5, and MiniMax-M2.5—publish no cutoff, so we overlay each one's earliest public release as a conservative later proxy (dotted line): a model provably cannot train on a task merged after its own release. Tasks to the left of the boundary (shaded) could have entered training; tasks to the right could not.

Contamination would raise pass rates on the shaded, potentially-seen tasks and lower them on the unseen ones to the right. Figure 12 shows no such effect: for every system, in both the *oneshot* and *gpu-tools* configurations, the pass rate is flat or rises into the unseen region. We therefore find no evidence that leaderboard standings are explained by training-data contamination; they reflect optimization ability rather than recall of the shipped gold solution.

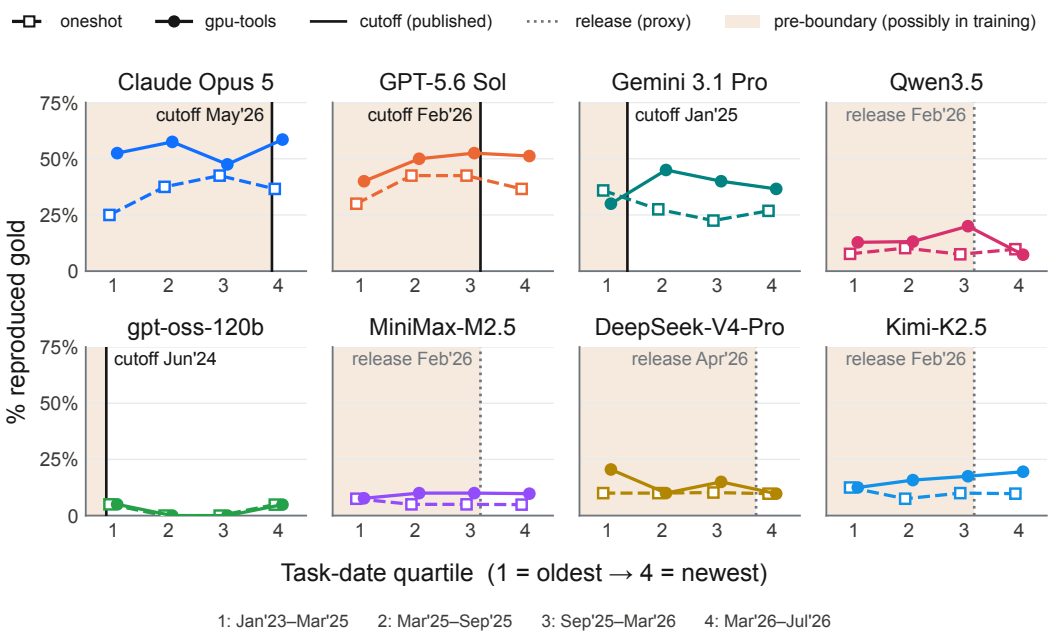


Figure 12: Pass rate by task date, per model, in the *oneshot* (open squares, dashed) and *gpu-tools* (filled circles, solid) configurations. Tasks are sorted by their source pull request's merge date and split into four equal-count bins numbered 1 (oldest) to 4 (newest); the x-axis is task-date rank, so the bins are equally spaced despite the recency skew (each bin's date span is given beneath the panels). Each model's boundary marks where it could plausibly have trained on the task: its published training-data cutoff (solid) or, where none is published, its earliest public release as a conservative proxy (dotted). Shading marks the pre-boundary region (tasks potentially seen in training). Pass rates do not fall on the post-boundary tasks a model could not have seen, indicating the leaderboard is not driven by contamination.

C.5 Cross-system speedup correlations

Figure 13 compares the task-level speedups achieved by pairs of systems within each interaction mode. Each correlation uses only tasks for which both systems produced correct solutions with measured speedups; failures, missing attempts, and correct but unmeasured solutions are omitted rather than assigned zero speedup. Consequently, sample sizes vary across system pairs, as reported in each cell. These correlations describe shared variation among successful measured solutions and should not be interpreted causally.

C.6 Performance by source implementation family

We examine whether optimization difficulty varies with the implementation family of the task source. Figure 14 reports each system's pass rate on PyTorch, Triton, and CUDA-family sources under the oneshot and gpu-tools settings. The CUDA family includes CUDA and related CUDA-based implementations such as CuPy and cuSolver; CuTe DSL is counted as its own target language.

We use the same system runs and pass definition as Section 5, grouping tasks only by source implementation family. We omit the small "Other" source group because it is too small to support a meaningful comparison.

For most systems, PyTorch source tasks have the lowest pass rates. Providing profiling feedback improves the results of many systems, but does not eliminate the differences between source families.

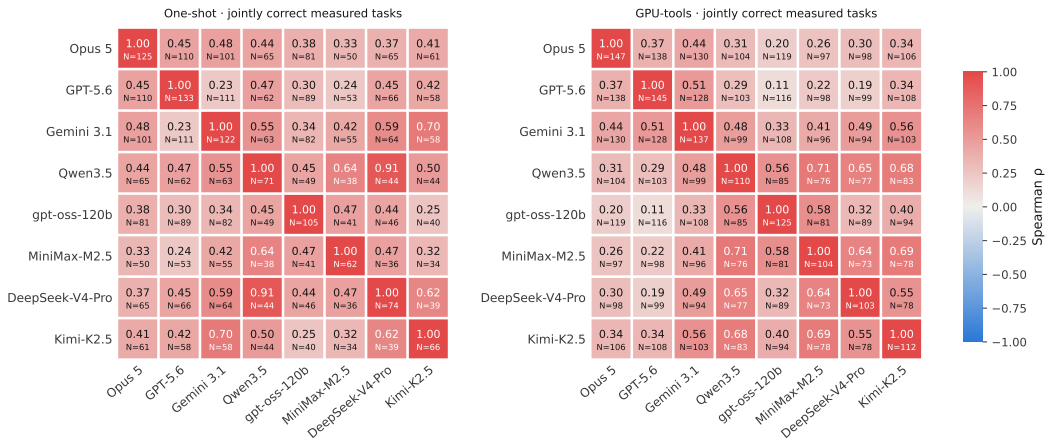
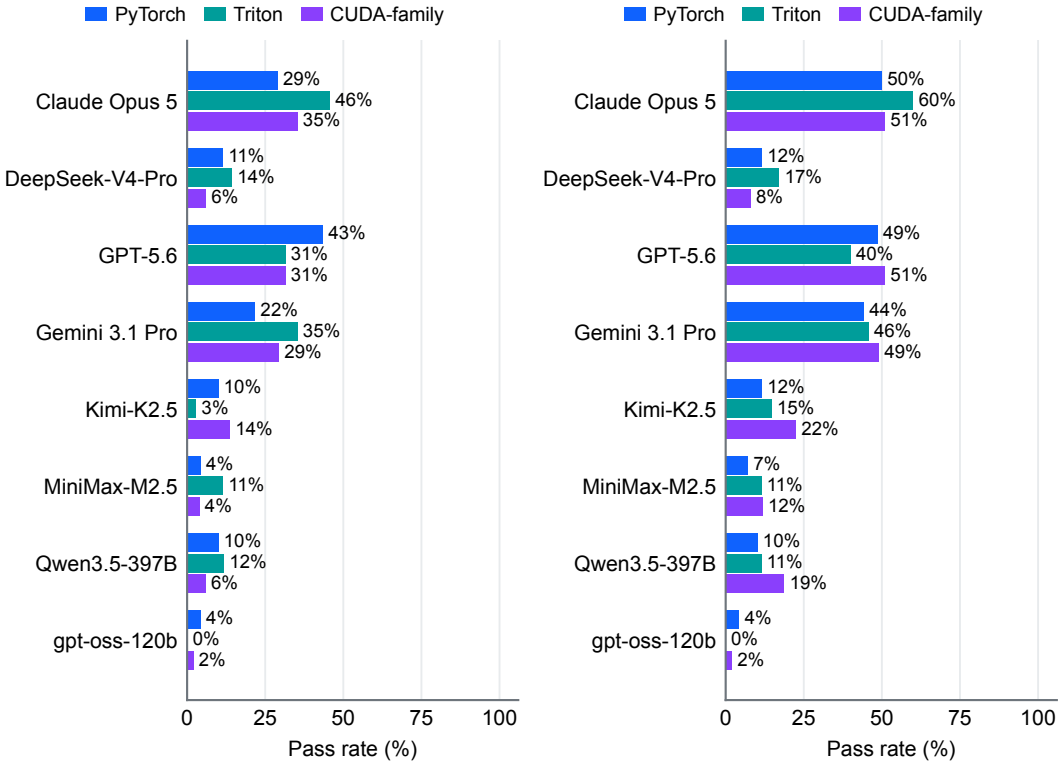


Figure 13: Pairwise Spearman correlations between system solution speedups in the *oneshot* and *gpu-tools* settings. Every cell reports the correlation and number of jointly correct, measured tasks for the corresponding system pair.



(a) One-shot **(b) GPU-tools**

Figure 14: Pass rate by source-implementation family and interaction mode. Bars report results for tasks with PyTorch, Triton, and CUDA-family sources. A task passes only if the solution is correct and matches or exceeds gold performance.

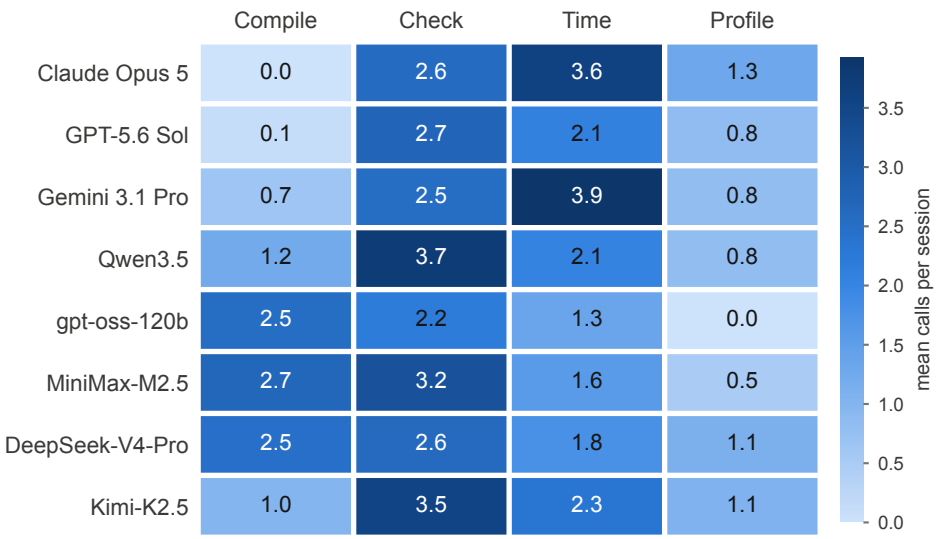


Figure 15: Verify-tool use during *gpu-tools*-mode generation, over the sessions behind the leaderboard's gpu-tools rows (Table 6). Each cell denotes the mean per-session calls a system makes to the different available GPU-verification tools —compile the kernel, check its correctness, time it, or profile it with Nsight Compute (`ncu`).

C.7 Tool-use analysis

Figure 15 breaks down how the *gpu-tools* mode systems spend their tool budget. It reports, per system, the mean per-session calls to the compile, correctness-check, timing, and Nsight Compute profiling tools, over the same sessions the leaderboard scores.

D Artifact and reproducibility

SWE-kernel-bench is released as standalone tasks that a third party can run on GPUs. Each task is scored by the single benchmark driver of §5.2, also released. The package ships with all benchmark tasks. For reproducibility, we also release the benchmark-construction code that extracts the tasks from PRs. Extracted tasks contain code from upstream projects under their own licenses and notice requirements. The release therefore carries, per task, the source repository, revision, license, and required attribution.