Agent Harnesses for GPU Kernel Optimization: A Taxonomy of Failure Modes and Design Implications

Alexandru Gherghescu[1], William Martin[1] and Rio Yokota[1]

[1]Institute of Science Tokyo, 2-12-1 Ookayama, Meguro-ku, Tokyo 152-8550, Japan

Abstract

Off-the-shelf state-of-the-art coding agents have largely cleared the basic-correctness bar for GPU kernel optimization, as measured through Codex and Claude on KernelBench, even under minimal scaffolding. What remains broken is not what they write but how they search. Both agents profile the kernels they cannot beat but rarely translate the readings into qualitatively different solutions, allocate search budget on rediscovering failed solutions on those same problems, and exit prematurely unless given a concrete baseline to beat during the optimization process. We argue that the bottleneck has shifted from model capability to harness design, and develop this argument empirically by running two off-the-shelf agents (Codex GPT-5.5 and Claude Opus 4.7) on the same workload under a deliberately minimal harness, characterizing recurring search-control failures from the resulting traces, and pairing each failure mode with a class of harness mechanism explored in recent specialized systems. The result is an empirically motivated roadmap for the design of the next generation of kernel-optimization harnesses.

Keywords

GPU kernel optimization, coding agents, agent harnesses

1. Introduction

At the forefront of today's coding revolution are agentic coding models: systems that can complete substantial work on their own. Agents have evolved from simple models that could complete a function or method body into full assistants capable of interacting with large codebases over extended trajectories, across millions of lines of code. Yet the central long-horizon problems remain unresolved: context management, search control, and degradation over time. These weaknesses are especially visible on open optimization tasks, where there is no single correct path and the agent must explore alternatives, run experiments, interpret evidence, and revise its approach without losing the thread of the work.

Kernels are the innermost layer of every GPU workload and are expensive to write by hand, which has made kernel generation a long-standing target for automation. GPU kernel optimization is, in particular, a useful stress test for long-horizon coding agents because success is measured, not merely judged. The model must generate kernels, run controlled experiments, interpret runtime or profiler feedback, and keep searching until it clears a strong baseline. Once the task becomes agentic, the central difficulty is no longer just *writing* kernel code. The harder problem is to maintain a search process that stays grounded, uses tools when needed, and keeps improving rather than prematurely declaring victory. Simply extending the wall-clock budget is not a substitute: without explicit search structure, agents hill-climb on a single solution, and additional time gets small improvements on a local optimum rather than qualitatively different approaches. That, in turn, makes harness design part of the research problem rather than mere experimental scaffolding [1, 2].

On the system side, earlier kernel generation work was largely a fixed workflow with an LLM as the code generating component; KernelBench's original setup [3], or the Kevin multi-turn RL system [4] both had the harness run code, check outputs, and feed back the next refinement turn through predefined

GeCoIn 2026: Generative Code Intelligence Workshop, co-located with the 35th International Joint Conference on Artificial Intelligence (IJCAI-ECAI 2026), August 15–17, 2026 — Bremen, Germany

alexghergh@rio.scrc.iir.isct.ac.jp (A. Gherghescu); will@rio.scrc.iir.isct.ac.jp (W. Martin); rioyokota@rio.scrc.iir.isct.ac.jp (R. Yokota)

https://www.rio.scrc.iir.isct.ac.jp/en/index.html (R. Yokota)

0009-0005-8460-4964 (A. Gherghescu); 0009-0008-5306-5373 (W. Martin); 0000-0001-7573-7873 (R. Yokota)

paths, with the LLM in predefined slots in a deterministic state machine. Current frontier coding systems such as Codex [1] and Claude [2] invert that arrangement: the LLM is in control of its own trajectory [5], decides when to use tools, and sometimes decomposes work into helper roles [6], while the harness exposes a constrained surface rather than dictating the next step. Goal drift remains real, but the central question shifts from *"can the model take another turn?"* to *"does the harness expose the right tools, state, and steering to keep long-horizon search productive"*?

This paper studies a practical question: *what control surfaces are still missing if one wants frontier coding agents to behave like persistent kernel engineers rather than short-horizon code assistants?* Because our intent is diagnostic rather than competitive, the minimal scaffolding around the agent is the experimental variable: we deliberately expect to underperform purpose-built systems and use that gap to surface what is still missing from off-the-shelf coding agents. The method does not yet implement the stronger control mechanisms seen in recent specialized systems, but it is already sufficient to expose recurring long-horizon failure patterns and organize them into a concrete taxonomy that can guide future harness design. We focus on H100 as the testbed and on two state-of-the-art coding agents at the time of writing: Codex GPT-5.5 [7] and Claude Opus 4.7 [8].

The paper makes three contributions. First, it reports as-run H100 results for Codex and Claude under a common harness. Second, it organizes the observed shortcomings into a taxonomy of practical search failures: weak stopping signals, local-search collapse on hard kernels, ineffective tool use, unstructured memory, and role entanglement. Third, it compares the current harness to stronger recent systems that add tree or population search [9, 10, 11], memory and skill libraries [12], planner–coder separation [9, 10], hardware-grounded guidance [13], and learned optimization priors [14], then turns those differences into a concrete ablation roadmap.

2. Experimental Setup

2.1. Task, benchmark, and testbed

The task is to write a low-level implementation (CUDA, Triton, or a device DSL) of a numerical operator that runs faster than the compiled PyTorch default and produces the same output under a numerical tolerance. We use KernelBench [3] as the reference benchmark. KernelBench established measured GPU kernel generation as a common testbed, and recent work such as CudaForge [15], STARK [9], CUDA Agent [14], KernelSkill [12], and KernelFoundry [11] also report results on it, which makes it the most useful reference point for positioning our harness. Problems are grouped in three levels: single operators (Level 1), small fused sequences (Level 2), and full architecture blocks (Level 3). Each problem ships a PyTorch reference model at a fixed shape; Level 1 problem 19, for instance, is `torch.relu` on a $4096 \times 393{,}216$ tensor.

All runs in this paper use NVIDIA H100 and bf16 baselines. Each problem is evaluated against eager PyTorch and default `torch.compile`. We study Codex GPT-5.5 and Claude Opus 4.7 on the full Level-1 subset of KernelBench (100 problems).

2.2. Method

Following auto-research designs for software-engineering tasks [16, 17], we wrap each KernelBench problem in a self-contained optimization environment. The agent receives a fixed PyTorch reference implementation and a narrow tool surface: timed evaluation, `ncu` profiling, and hosted web search limited to NVIDIA documentation. The agent chooses when to generate, time, or profile kernels; correctness and performance are determined by the method's measurements, not by the agent's self-reports. The agent is also given a small set of steering documents. These cover the optimization task and its success criteria, an autonomy contract for tool use, H100 hardware facts and tuning guidance, and a live-view document updated with remaining budget and best-so-far performance. Each problem comes with two explicit baseline runtimes to beat: eager PyTorch and default `torch.compile`.

Table 1
Results on H100, bf16 baselines, 180-minute per-problem budget, KernelBench Level 1 100-problems set. Speedups are relative to default `torch.compile`. Total input tokens include cached input; estimated cost applies separate per-rate charges to cached and uncached input.

Agent	Correct	> compile	Median ×	Geomean ×	Input (M)	Output (M)	Est. API cost
Codex	99.0%	73.0%	1.06×	1.18×	826.68	8.59	$835.76
Claude	100.0%	67.0%	1.08×	1.04×	509.55	18.60	$855.05

Every problem is given the same fixed time budget of 180 wall-clock minutes on a dedicated H100 GPU. Within that budget the agent decides when to generate kernels, when to time them, when to profile them, and when to stop; the method only enforces the budget cap and the tool surface, not the search policy. Both agents run at `xhigh` reasoning-effort.

2.3. Measurement policy

KernelBench's headline metric is `fast_p`: the fraction of problems with a correct kernel at least $p\times$ faster than a chosen baseline. The primary outcome in this paper is therefore *best-correct performance*: for each problem, we record whether the run produced any correct kernel, and whether any correct kernel beat eager PyTorch and default `torch.compile` respectively. We additionally report medians and geometric means over per-problem best-correct speedups, summarizing *how much faster* a kernel gets once the problem is solved.

More precisely, let $\mathcal{C}$ denote the set of problems for which the run produced at least one correct kernel. For each $i \in \mathcal{C}$, let t_i^{best} be the runtime of the fastest correct kernel found by the run, and let $b_i^{(B)}$ be the runtime of baseline $B \in \{\text{eager}, \text{compile}\}$. The reported per-problem speedup against baseline B is then

$$s_i^{(B)} = \frac{b_i^{(B)}}{t_i^{\text{best}}}.$$

The two aggregate statistics are:

$$\text{median}^{(B)} = \text{median}\{s_i^{(B)} : i \in \mathcal{C}\}, \qquad \text{geomean}^{(B)} = \left(\prod_{i\in\mathcal{C}} s_i^{(B)}\right)^{1/|\mathcal{C}|}$$

In other words, both aggregates are computed over the per-problem *best-correct* speedups, not over all attempted kernels. Problems with no correct kernel are excluded from these speedup aggregates.

We also surface wall-clock effort, candidate kernel evaluations, documentation-search counts, token totals, and API cost. Codex cost is reconstructed from recorded token usage in traced runs using vendor API prices accessed on May 12, 2026 [18], while Claude reports a billed cost in run traces, but this coincides with its advertised API prices [19]. These cost figures are therefore useful operational indicators of true optimization cost.

3. Main Results

Table 1 summarizes headline outcomes and Figure 1 per-agent search effort; a *candidate kernel evaluation* is one harness-recorded candidate check with measured compile, correctness, and timing outcomes. Both agents, under the same minimal harness, clear correctness on most KernelBench Level 1 problems (Codex 99 of 100; Claude 100 of 100) and beat default `torch.compile` on a majority (73.0% and 67.0% respectively, with compile-relative geomean speedups of 1.18× and 1.04×). This shows that off-the-shelf frontier coding agents are, at minimum, capable of producing correct, faster-than-baseline CUDA kernels.

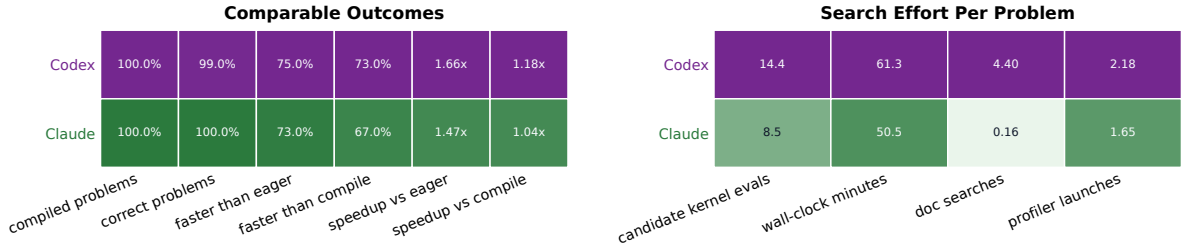

Figure 1: Per-agent overview. Left panel: outcome metrics (rates as % of problems; geomean speedups). Right panel: mean per-problem search effort. Cell color normalized per column.

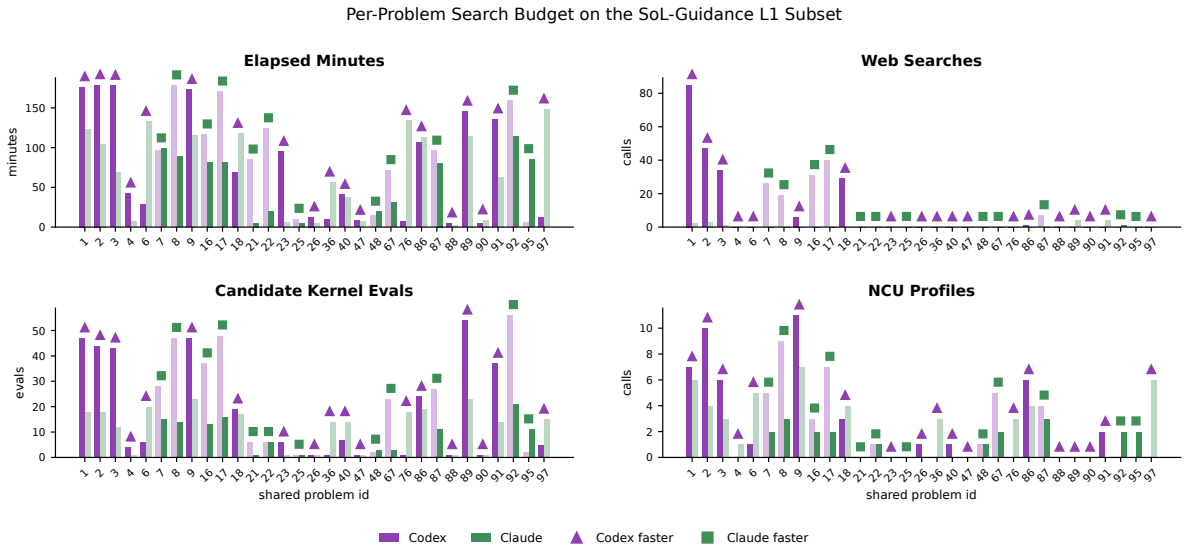

Figure 2: Per-problem search-effort comparison across elapsed minutes, documentation searches, candidate kernel evaluations, and `ncu` profile calls, restricted to the 31 Level 1 problems in the μCUTLASS + SOL-guidance subset [13], dominated by compute-bound matmul and convolution variants. Marks above the bar pairs show the faster best-correct winner.

The two agents arrive at these outcomes from somewhat different working styles. Codex pushes compile-relative speed slightly harder (73.0% vs 67.0% faster-than-`torch.compile`, 1.18× vs 1.04× geomean), while Claude wins more direct head-to-head comparisons (54 vs 46). Total API cost is comparable, but the token mix differs, with Codex consuming roughly 1.6× the input tokens while Claude generates more than twice the output, plausibly reflecting each agent's input and output token distributions from training rather than anything kernel-specific. On the μCUTLASS + SoL-guidance [13] subset of compute-bound L1 problems (Figure 2), both agents push noticeably harder than on the rest of the workload, spending around 1.6× the elapsed time and running 50–75% more candidate kernel evaluations per problem. Both can sustain a long search when the problem demands it.

However, the headline numbers hide a disconnect between *finding correct kernels* and *making them faster than the baseline*: a meaningful fraction of correct kernels still trail default `torch.compile` (Codex 73/99 beat compile, Claude 67/100). The harness is strong enough to support initial discovery, but not yet strong enough to reliably steer search toward high-quality kernels: several hard problem runs consume the full 180-minute budget without managing to beat baselines. This points toward a need to refine the search process on the harder problems, showing that once a problem gets difficult, the agents lack a clear optimization path. What's needed is specialized harness machinery to keep the agents productive across long-horizon tasks.

The gap to stronger specialized systems points in the same direction. STARK [9] reports 78.6% faster-than-`torch.compile` kernels on its representative Level 1 subset, with 2.37× average compile-relative

Table 2
Failure modes observed in our runs, with the evidence anchor for each. Per-row design implications are developed in Section 5.

Failure mode	Observable symptom	Evidence anchor
Weak stopping signal	Without a given goal to beat, models tend to give up early after only partially exploring viable alternative optimization paths.	On problem 40 (LayerNorm), both agents produce correct kernels but stop at ∼40 min with >140 min unused, at 0.28× and 0.22× compile respectively. More broadly, 3 Codex / 11 Claude problems stop before 90 min despite never beating compile, and 86% / 81% of compile-beating runs record no further candidate evaluations after the first winning kernel (Appendix A).
Optimization-space exploration	On the harder compute-bound problems, the agent commits to one strategy class early and then iterates locally within it instead of exploring qualitatively different approaches.	E.g., on problem 11 (4D tensor matrix multiplication) Codex plateaus around attempt 26 (∼0.36× eager) and spends the remaining 26 attempts hovering at near-identical runtimes (2.80–2.90 ms) without breaking out of the local optimum.
Tool-use policy	Tools are available and frequently invoked on hard problems, but their outputs rarely produce qualitatively different kernels; web search is under-used by Claude.	On problems where neither agent beats any baseline (Codex 25/100, Claude 26/100), profiler use is near-universal (Codex 25/25 = 100%, Claude 25/26 = 96%), but web search is not even: Codex 18/25 (72%), Claude only 5/26 (19%).
Context management	The agent doesn't retain depth of trajectory history: prior kernels and the lessons they yielded fall out of working context as the run lengthens.	E.g., on problem 11 (4D tensor matrix multiplication), Codex regenerates a byte-identical kernel at attempts 37, 39, and 43 after intervening alternatives failed to improve — trajectory state isn't externally stored, so prior work is rediscovered rather than retrieved.
Role entanglement	One agent is asked to plan, code, judge, and assess stopping criteria.	Sub-agent use is heavy but narrow: 1651 Codex and 966 Claude spawns, essentially all to the harness-provided `runner` and `profiler` tool helpers (>99% for Claude by subagent-type metadata; all identified spawn prompts for Codex); planning, coding, and judgment stay in the main agent's context and accumulate as the run lengthens.

speedup, via plan / code / debug role separation. CUDA Agent [14], by contrast, is notable here not because it saturates correctness, but because it pushes the stronger metric: on Levels 1–3 it reports 97% / 100% / 90% faster-than-`torch.compile` rates, with compile-relative geomean speedups of 1.87× / 2.80× / 1.52× via an RL-trained search prior plus a skill-augmented environment. KernelSkill [12] reaches 100% problem-level correctness across Levels 1–3 with dual-level memory, and CudaForge [15] reports 97.6% correctness and 1.68× eager-relative speedup via a coder–judge workflow. These numbers come from different paper-specific subsets, baselines, and metrics, so the comparison is informative rather than head-to-head; this gap shows what our method still lacks: explicit search controllers, memory, role separation, and learned priors.

4. Practical Failure Modes in Long-Horizon Kernel Search

The harder question, then, is not whether the agent can produce correct CUDA, but how far it can push a correct kernel toward high performance. In our runs, the dominant failures are not in syntax or compilation but in agent search control. Table 2 names the five modes, with the evidence anchor for each; the paragraphs below take them in turn, and Section 5 discusses what each implies for future harness design.

Baselines help, but only at the first layer. Explicit baselines matter; without them, agents tend to stop at first correctness. With them, agents push to beat the baseline. But the baseline is a binary signal: once it is beaten, both agents stop almost immediately, even when more performance may be available — 86% of Codex and 81% of Claude compile-beating runs record no further candidate evaluations after the first winning kernel. The opposite failure shows up when the baseline is not beaten: Claude gives up before 90 min on 11 such problems, whereas Codex more often runs to the full 180-min budget (Appendix A).

Hard problems expose under-exploration, not inactivity. The exploration failure is most visible on large compute-bound problems. The agent is not inactive — it compiles, measures, keeps trying — but search keeps working on the current trajectory rather than broadening to qualitatively different strategies. Working longer and searching better are not the same.

Tools are used but rarely unblock the agent. On problems where neither agent beats any baseline, profiling is consulted near-universally (Codex 100%, Claude 96%), but the readings rarely translate into qualitatively different kernels. Web search shows a sharper difference: Codex consults documentation on most losing problems (72%), while Claude almost never does (19%). The failure is tool non-use, but it's also a missing loop between a profiler reading and a different algorithmic approach.

Trajectory state isn't retained as the run lengthens. What falls out of context is not memory of facts but memory of *search trajectory*: failed branches, exhausted directions, and reusable tactics. When the agent re-encounters a similar problem state mid-run, it tends to rediscover failed approaches rather than retrieve them from prior work.

An agent used for both exploration and precision is over-tasked. The same agent currently supports planning, implementation, measurement interpretation, and stopping, four cognitive postures with different failure modes. Earlier role context (system prompt, earlier outputs) in the run still conditions current output, due to the attention mechanism, so a model tasked with being a good coder cannot reliably switch to being a skeptical reviewer just because it was asked to.

5. Design Implications for Stronger Harnesses

The strongest recent systems do not merely "prompt better"; they add structure around the model. A consistent move is to push state *out of the conversation*: search branches into trees or graphs, reusable tactics into skill libraries, bottleneck knowledge into profile summaries, stopping criteria into explicit target levels, tool-use policy into controllers, so that hard search does not depend only on the model's context window. A second move is to separate exploration, implementation, and judgment into typed roles rather than collapsing them into one agent. Because Codex and Claude Code are closed frontier agents, the immediate roadmap below lies on the harness side; agent-side improvements such as learned priors and reinforcement learning are deferred to the closing paragraph.

Baselines, headroom, stopping. The weak stopping signal is structural: a single binary baseline gives the agent nothing to push against once it is beaten. Keep eager and `torch.compile` as the first baselines, but stop treating them as the final target; once a kernel beats them, switch to a hardware-specific headroom target (e.g. SoL-style roofline views [13, 20, 21, 22]), so the agent knows how much plausible room is left for the kernel. The same mechanism should also control time budget: continue to give the agent new budget only when a strictly better kernel appears, and force a pivot or stop once that extra budget is exhausted [23]. Search should stay open-ended only while there is reason to believe it is paying off.

External search state. Optimization-space under-exploration is a symptom of a weak search representation: an agent's optimization history is only what remains inside the context window. A plausible improvement is for the harness to instead represent each evaluated candidate as a node in an external graph, as *parent branch*, *tactic*, *measured runtime*, *compile/correctness status*, *profiler summary*, and require the planner to issue an explicit *deepen search*, *pivot strategy*, or *stop* decision at each checkpoint. K-Search [10] is the clearest example because it treats non-monotonic branch planning as a first-class problem; STARK [9], EvoEngineer [24], KernelFoundry [11], and Kernel-Smith [25] all keep alternative branches alive instead of relying on one serial trajectory.

Tool-use policy. The tool-use failure is that availability does not necessarily lead to improvements. The harness should specify *when* the model should seek external evidence: after several correct but non-improving kernels, require profiling or skill retrieval before further local tweaking; after repeated bottleneck uncertainty, require web lookup before more guessing. This is the actionable takeaway from budget-aware tool policies such as BATS [23] and structured tool-planning work such as ToolTree [26] and Lost in the Maze [27]: a stronger policy is better than a bare tool catalog.

Memory and reusable skills. Trajectory state falls out of context because nothing external stores it. The solution is a dual-level memory system in the style of KernelSkill [12], extended with reusable-memory ideas from Skill-Pro [28] and KernelBlaster [29]. Short-term memory should record failed rewrites, exhausted parameter regions, and disproven bottleneck explanations for the current problem so the agent does not keep replaying failed solutions. Long-term memory should store reusable *skill cards* that transfer across runs: problem archetype, preconditions, optimization mechanism, a parameterized template, expected profiler or runtime signature, known failure cases, and *negative scope* (when *not* to retrieve the skill). Voyager [30] introduced this idea; recent frontier coding systems now expose comparable skill mechanisms at inference time [31, 32, 33], which makes the pattern directly available in our setting.

Planner–coder–judge split. Role entanglement needs a structural split rather than stronger prompting; the separation must be explicit, by using multiple agents (agents with different context) [34, 35]. K-Search [10] makes the strongest concrete argument, since complex kernels often require multi-step structural changes that might look bad early if planning and implementation collapse into one greedy loop, instead of keeping multiple optimization threads open at the same time and combining ideas; STARK [9] and CudaForge [15] push the same separation. A practical split is planner–coder–judge: the planner chooses branches and decides when to pivot or deepen, the coder implements one branch under a strict local objective, and the judge interprets wrapper and profiler outputs to decide whether the branch improved, stalled, or invalidated an assumption. The handoff between roles should be compact: branch hypothesis, expected bottleneck, allowed tactic set, required measurements, stop condition.

Other directions. Two families of work are outside the immediate roadmap identified above. *Persistent experiment loops* (AlphaEvolve [36], ASI-Evolve [37]) treat each run's experiments and takeaways as the starting context for the next; we partially capture this idea in the long-term skill library proposed above. *Learned optimization priors* (Kevin [4], ConCuR/KernelCoder [38], CUDA-L1 [39], CUDA Agent [14]) train stronger models through RL, using feedback from local trajectories; the broader reasoning literature suggests RL is most useful when embedded in stronger search structure rather than treated as a standalone fix [40, 41, 42, 43, 44]. This, however, is agent-side, and so does not directly apply to closed frontier LLMs, but the structured traces these runs produce could in principle support distilling knowledge of kernel execution and successful trajectories into smaller models.

Table 3
Suggested harness improvements for more performant coding agents, directly matching identified failures.

Roadmap item	Immediate implementation	Expected effect
Baselines, headroom, and stopping	Keep eager and `torch.compile` as the first baselines to beat, add a kernel-specific hardware-headroom target, and use a rolling continuation budget that resets only after real runtime improvements.	Model keeps optimizing after the first win, but stops spending once further search stops producing evidence-backed gains.
External search state	Represent each evaluated candidate as a graph node with parent, tactic, result, and profiler summary; let the planner deepen or pivot over explicit branches.	Model retains and revisits past optimization paths instead of losing them inside one conversation.
Tool-use policy	Require profiler, web, or skill lookup once repeated non-improving attempts or uncertain bottlenecks show the model has hit a local wall.	Model seeks profiler or web evidence when it gets stuck instead of continuing unguided.
Memory and reusable skills	Add short-term anti-repeat memory plus long-term skill cards with preconditions, expected signatures, and negative scope.	Model avoids repeating failed tactics and reuses prior tactics when the current conditions match.
Planner–coder–judge split	Separate branch selection, branch implementation, and result interpretation into typed handoffs.	Model explores, implements, and evaluates separately instead of collapsing into one greedy loop.

6. Limitations

Scope. The runs cover KernelBench Level 1 only, on a single H100 device, at a uniform 180-minute budget, with one trajectory per problem. Level 2/3 problems, alternative hardware, and multi-seed variance estimates are out of scope; at closed-API prices, multi-seed runs would drive costs up several times.

Precision and workload. Baselines are bf16 only. Production kernels often care about fp16 or TF32 as well, and fp32 and fp64 still matter in classical HPC; backward-pass kernels and broader memory-bound operator families are also not exercised here. Recent multi-precision and cross-platform work [11, 22] suggests agent behavior can shift substantially once shapes, hardware, or precision vary.

Cost accounting. Codex cost is reconstructed from per-token rates; Claude uses trace-reported billed USD. Our reported costs are estimated only, though vendor list prices match the trace numbers within rounding.

Cross-system comparisons. Numbers from recent specialized systems were obtained on different hardware, backends, benchmark subsets, and metric definitions. The gap discussion in this paper is therefore informative only, not head-to-head.

Moving target. Findings are tied to Codex GPT-5.5 and Claude Opus 4.7 as of May 2026. Specific numbers will shift with model updates. We expect the taxonomy categories themselves to outlive these specific versions, but that claim is not validated here.

7. Conclusion

Recent model improvements have made substantially more ambitious harness projects possible. Frontier coding agents can already sustain longer trajectories, use tools, and produce correct kernels that often beat strong software baselines. That is no longer the main open question.

The harder question is how to give these agents the right external structure, and how to work around their current limitations once the easy wins are gone. In this study, we show that the main bottlenecks are not basic correct code generation anymore, but search drift, weak exploration, tool use, and lack of explicit search state. The practical path forward is therefore not more prompt changes, it's building better controllers around the model: explicit branch state, reusable memory, stronger tool-use policy, and clearer separation between planning, implementation, and evaluation. Coupled with stronger foundational models, this fundamentally shifts harness development efforts; systems look less like *"can the model write CUDA?"* and more like *"can the harness keep a productive experiment loop alive?"*.

Declaration on Generative AI

During the preparation of this work, the author(s) used Anthropic Claude and OpenAI Codex in order to: paraphrase, reword, review, fix grammar. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content. The same systems are also the subject of study in this paper; outputs they produced as agents under evaluation appear in the experimental data analyzed throughout.

Acknowledgments

This work is partially supported by MEXT as "Project for Establishment of a Center for Advanced HPC-AI Development Support".

References

[1] OpenAI, Run long horizon tasks with codex, https://developers.openai.com/blog/run-long-horizon-tasks-with-codex/, 2026. OpenAI Developers blog.

[2] Anthropic, Effective harnesses for long-running agents, https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents, 2026. Anthropic engineering blog.

[3] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, A. 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*, PMLR, 2025, pp. 47356–47415. URL: https://proceedings.mlr.press/v267/ouyang25a.html.

[4] C. Baronio, P. Marsella, B. Pan, S. Guo, S. Alberti, Kevin: Multi-turn rl for generating cuda kernels, 2025. URL: https://arxiv.org/abs/2507.11948. arXiv:2507.11948.

[5] Anthropic, Long-running tasks, https://www.anthropic.com/research/long-running-tasks, 2026. Anthropic research.

[6] Anthropic, Building a c compiler with a team of parallel claudes, https://www.anthropic.com/engineering/building-c-compiler, 2026. Anthropic engineering blog.

[7] OpenAI, Introducing gpt-5.5, https://openai.com/index/introducing-gpt-5-5/, 2026. OpenAI blog.

[8] Anthropic, Introducing claude opus 4.7, https://www.anthropic.com/news/claude-opus-4-7, 2026. Anthropic news.

[9] Y. Cao, T. Li, Y. Zhang, P. Zhang, S. Chen, Y. Luo, D. He, M. Yang, S. Jiang, Stark: Strategic team of agents for refining kernels, 2025. URL: https://arxiv.org/abs/2510.16996. arXiv:2510.16996.

[10] S. Cao, Z. Mao, J. E. Gonzalez, I. Stoica, K-search: Llm kernel generation via co-evolving intrinsic world model, 2026. URL: https://arxiv.org/abs/2602.19128. arXiv:2602.19128.

[11] N. Wiedemann, Q. Leboutet, M. Paulitsch, D. Wofk, B. Ummenhofer, Kernelfoundry: Hardware-aware evolutionary gpu kernel optimization, 2026. URL: https://arxiv.org/abs/2603.12440. arXiv:2603.12440.

[12] Q. Sun, J. Han, T. Li, Z. Tang, S. Chen, F. Yang, A. Liu, X. Liu, Y. Liu, Kernelskill: A multi-agent framework for gpu kernel optimization, 2026. URL: https://arxiv.org/abs/2603.10085. arXiv:2603.10085.

[13] S. K. S. Hari, V. Balaji, S. Damani, Q. Huang, C. Kozyrakis, Improving efficiency of gpu kernel optimization agents using a domain-specific language and speed-of-light guidance, 2026. URL: https://arxiv.org/abs/2603.29010. arXiv:2603.29010.

[14] X. Wang, C. Tao, Y. Chen, Y. Lu, G. Luo, C. Lyu, F. Meng, J. Zhou, Y. Zheng, Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation, 2026. URL: https://arxiv.org/abs/2602.24286. arXiv:2602.24286.

[15] Z. Zhang, R. Wang, S. Li, Y. Luo, M. Hong, C. Ding, Cudaforge: An agent framework with hardware feedback for cuda kernel optimization, 2025. URL: https://arxiv.org/abs/2511.01884. arXiv:2511.01884.

[16] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, O. Press, SWE-agent: Agent-computer interfaces enable automated software engineering, in: Advances in Neural Information Processing Systems 38 (NeurIPS), 2024. URL: https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

[17] A. Karpathy, karpathy/autoresearch, https://github.com/karpathy/autoresearch, 2026. GitHub repository.

[18] OpenAI, Openai api pricing, https://openai.com/api/pricing/, 2026. Accessed May 12, 2026.

[19] Anthropic, Anthropic api pricing, https://docs.anthropic.com/en/docs/about-claude/pricing, 2026. Accessed May 12, 2026.

[20] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Communications of the ACM 52 (2009) 65–76. doi:10.1145/1498765.1498785.

[21] J. Jaber, O. Jaber, Autokernel: Autonomous gpu kernel optimization via iterative agent-driven search, 2026. URL: https://arxiv.org/abs/2603.21331. arXiv:2603.21331.

[22] Q. Qu, Y. Sui, Y. Sun, R. Chen, X. Zhang, Y. Zhang, H. Wang, G. Lan, N. Zhang, A two-stage gpu kernel tuner combining semantic refactoring and search-based optimization, 2026. URL: https://arxiv.org/abs/2601.12698. arXiv:2601.12698.

[23] T. Liu, Z. Wang, J. Miao, I.-H. Hsu, J. Yan, J. Chen, R. Han, F. Xu, Y. Chen, K. Jiang, S. Daruki, Y. Liang, W. Y. Wang, T. Pfister, C.-Y. Lee, Budget-aware tool-use enables effective agent scaling, 2025. URL: https://arxiv.org/abs/2511.17006. arXiv:2511.17006.

[24] P. Guo, C. Zhu, S. Chen, F. Liu, X. Lin, Z. Lu, Q. Zhang, Evoengineer: Mastering automated cuda kernel code evolution with large language models, 2025. URL: https://arxiv.org/abs/2510.03760. arXiv:2510.03760.

[25] H. Du, Q. Ge, J. Hu, A. Yang, et al., Kernel-smith: A unified recipe for evolutionary kernel optimization, 2026. URL: https://arxiv.org/abs/2603.28342. arXiv:2603.28342.

[26] S. Yang, S. C. Han, Y. Ding, S. Wang, E. Hovy, Tooltree: Efficient llm tool planning via dual-feedback monte carlo tree search and bidirectional pruning, in: International Conference on Learning Representations, 2026. URL: https://openreview.net/forum?id=Ef5O9gNNLE.

[27] H. Yen, A. Paranjape, M. Xia, T. Venkatesh, J. Hessel, D. Chen, Y. Zhang, Lost in the maze: Overcoming context limitations in long-horizon agentic search, 2025. URL: https://arxiv.org/abs/2510.18939. arXiv:2510.18939.

[28] Q. Mi, Z. Ma, M. Yang, H. Li, Y. Wang, H. Zhang, J. Wang, Skill-pro: Learning reusable skills from experience via non-parametric ppo for llm agents, 2026. URL: https://arxiv.org/abs/2602.01869. arXiv:2602.01869.

[29] K. Dong, et al., Kernelblaster: Continual cross-task cuda optimization with persistent knowledge and textual gradients, 2026. URL: https://arxiv.org/abs/2602.14293. arXiv:2602.14293.

[30] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, A. Anandkumar, Voyager: An open-ended embodied agent with large language models, Transactions on Machine Learning Research (2024). URL: https://openreview.net/forum?id=ehfRiF0R3a.

[31] OpenAI, Agent skills – codex, https://developers.openai.com/codex/skills/, 2026. Accessed May 12, 2026.

[32] OpenAI, Shell + skills + compaction: Tips for long-running agents that do real work, https://developers.openai.com/blog/skills-shell-tips/, 2026. Accessed May 12, 2026.

[33] Anthropic, Extend claude with skills, https://docs.anthropic.com/en/docs/claude-code/skills, 2026. Accessed May 12, 2026.
[34] M. Cemri, M. Z. Pan, S. Yang, L. A. Agrawal, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, D. Klein, K. Ramchandran, M. Zaharia, J. E. Gonzalez, I. Stoica, Why do multi-agent llm systems fail?, in: Advances in Neural Information Processing Systems 38: Datasets and Benchmarks Track (NeurIPS), 2025. URL: https://openreview.net/forum?id=fAjbYBmonr, spotlight presentation.
[35] Z. Lyu, Y. Wang, H. Wang, J. Xu, T. Lan, et al., Understanding and bridging the planner-coder gap: A systematic study on the robustness of multi-agent systems for code generation, 2025. URL: https://arxiv.org/abs/2510.10460. arXiv:2510.10460.
[36] A. Novikov, N. Vu, M. Eisenberger, E. Dupont, et al., Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL: https://arxiv.org/abs/2506.13131. arXiv:2506.13131.
[37] W. Xu, T. Mi, Y. Liu, Y. Nan, Z. Zhou, L. Ye, L. Zhang, Y. Qiao, P. Liu, Asi-evolve: Ai accelerates ai, 2026. URL: https://arxiv.org/abs/2603.29640. arXiv:2603.29640.
[38] L. Kong, J. Wei, H. Shen, H. Wang, Concur: Conciseness makes state-of-the-art kernel generation, 2025. URL: https://arxiv.org/abs/2510.07356. arXiv:2510.07356.
[39] X. Li, X. Sun, A. Wang, J. Li, C. Shum, Cuda-l1: Improving cuda optimization via contrastive reinforcement learning, 2025. URL: https://arxiv.org/abs/2507.14111. arXiv:2507.14111.
[40] DeepSeek-AI Team, Deepseek-r1 incentivizes reasoning in llms through reinforcement learning, Nature 645 (2025) 633–638. URL: https://www.nature.com/articles/s41586-025-09422-z. doi:10.1038/s41586-025-09422-z.
[41] M. Liu, S. Diao, X. Lu, J. Hu, X. Dong, Y. Choi, J. Kautz, Y. Dong, Prorl: Prolonged reinforcement learning expands reasoning boundaries in large language models, 2025. URL: https://arxiv.org/abs/2505.24864. arXiv:2505.24864.
[42] F. Wu, et al., Deepsearch: Integrating monte carlo tree search into rlvr training for improved exploration, 2025. URL: https://arxiv.org/abs/2509.25454. arXiv:2509.25454.
[43] W. Tan, F. Parascandolo, E. Sangineto, J. Ju, Z. Luo, Q. Cao, R. Cucchiara, R. Song, J. Luan, Restoring exploration after post-training: Latent exploration decoding for large reasoning models, 2026. URL: https://arxiv.org/abs/2602.01698. arXiv:2602.01698.
[44] P. Mayilvahanan, R. Dominguez-Olmedo, T. Wiedemer, W. Brendel, Math-beyond: A benchmark for rl to expand beyond the base model, 2025. URL: https://arxiv.org/abs/2510.11653. arXiv:2510.11653.

A. Stopping Behavior

Table 4 lists every Level 1 problem where the agent stopped with at least 60 minutes of budget remaining (elapsed $<$ 120 min out of a 180 min budget), produced a correct solution, but did not beat the `torch.compile` baseline. In each of the 38 cases the agent chose to stop on its own, calling `complete_problem` with a best candidate still slower than compile; the budget cap never triggered.

Problem 40 (LayerNorm, $16 \times 64 \times 256 \times 256$). Both agents produced correct custom kernels but called `complete_problem` well before their budget ran out. Claude stopped at 37.8 min after 9 attempts (142 min remaining); Codex at 40.9 min after 5 attempts (139 min remaining). Claude's best candidate was 1.15 ms against a compile baseline of 0.254 ms (0.22×); Codex reached 0.916 ms (0.28×). Both kernels beat the eager baseline of 11.5 ms comfortably but were far from `torch.compile`. Neither agent continued searching despite having both a correct starting point and over two hours of unused budget.

Early-stop pattern among compile-beating runs. The converse pattern is equally pronounced. Counting candidate evaluations after the first kernel that beat compile: 61 of 71 Codex compile-beating runs with complete per-attempt records (86%) and 54 of 67 Claude runs (81%) recorded none at all,

Table 4
Problems with $\geq$60 min remaining at stop, a correct solution, but no compile-baseline win. Speedup vs. compile is $t_{\text{compile}}/t_{\text{best}}$; values below 1.0× mean the agent's best kernel is slower than `torch.compile`.

Problem	Agent	Elapsed (min)	Attempts	Best (ms)	Compile (ms)	Speedup vs. compile
40	Claude	37.8	9	1.150	0.254	0.22×
91	Claude	63.1	14	10.200	8.590	0.84×
13	Claude	63.4	13	0.604	0.280	0.46×
3	Claude	69.2	12	1.520	0.597	0.39×
11	Claude	75.8	18	4.430	1.160	0.26×
93	Claude	81.3	17	6.430	3.090	0.48×
17	Claude	81.6	16	0.704	0.271	0.38×
16	Claude	82.0	13	0.665	0.271	0.41×
69	Claude	82.1	10	5.310	1.820	0.34×
10	Claude	85.9	21	0.210	0.130	0.62×
8	Claude	89.2	14	1.850	0.698	0.38×
62	Claude	93.7	13	14.100	1.830	0.13×
56	Claude	96.1	13	6.110	2.510	0.41×
7	Claude	98.9	15	1.150	0.871	0.76×
2	Claude	104.5	18	0.551	0.274	0.50×
55	Claude	110.0	17	15.300	3.060	0.20×
58	Claude	112.1	20	4.960	1.230	0.25×
86	Claude	113.4	19	3.150	2.580	0.82×
68	Claude	113.5	16	25.200	7.490	0.30×
92	Claude	113.9	21	6.450	2.610	0.40×
70	Claude	114.1	24	10.800	7.650	0.71×
89	Claude	114.5	23	7.390	2.930	0.40×
9	Claude	115.8	23	0.906	0.862	0.95×
65	Claude	116.2	17	3.360	1.470	0.44×
18	Claude	117.6	17	0.798	0.272	0.34×
40	Codex	40.9	5	0.916	0.254	0.28×
18	Codex	69.4	18	2.070	0.272	0.13×
64	Codex	88.5	25	4.830	3.890	0.81×
87	Codex	96.0	27	7.150	6.890	0.96×
78	Codex	96.0	29	3.430	2.070	0.60×
7	Codex	96.7	28	1.980	0.871	0.44×
55	Codex	97.0	20	9.220	3.060	0.33×
56	Codex	99.5	27	7.900	2.510	0.32×
14	Codex	107.6	30	0.335	0.301	0.90×
13	Codex	110.6	31	1.700	0.280	0.16×
80	Codex	111.8	28	5.740	1.800	0.31×
16	Codex	116.4	37	1.410	0.271	0.19×
10	Codex	117.4	35	0.321	0.130	0.40×

and 97% / 88% recorded at most two. 58 Codex and 57 Claude problems beat compile within 60 min of elapsed time. Both patterns point at the same cause: the baseline is a binary stop signal, so a single first win ends the search regardless of remaining budget or headroom above the compile threshold.