REWARD HACKING IN SELF-IMPROVING CODE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recursive self-improvement (RSI) systems that iteratively optimize code with execution feedback are increasingly used for automated algorithm and systems optimization. A central failure mode is *reward hacking*: the agent improves a cheap proxy metric without improving or even harming the true objective under more realistic evaluation. We present a large-scale quantitative study of reward hacking in iterative code optimization by agents across two settings: GPU kernel optimization and algorithmic optimization. Across three frontier models and five agent configurations, we analyze thousands of agent trajectories under a setting where we have access to a held set of tasks (real tasks) for code evaluation while the agent only has access to a public set of evaluations (proxy tasks). Reward hacking is pervasive: among our experiments, 73.8% of Kernel-Bench optimizations and 46.8% of ALE-Bench optimizations exhibit proxy gains without gains in the real tasks. A temporal analysis shows the proxy-reality gap widens with optimization steps, going from 10 steps to 100 steps of optimization, percentage of reward hacking rises 31.4% from 26.4% to 57.8%. With this quantitative evaluation framework, we are able to evaluate techniques that may prevent reward hacking such as *retrospection*, a lightweight self-critique intervention triggered either probabilistically or on significant proxy metrics jumps. Retrospection reduces Kernel-Bench hacking by $\sim$17–19 points in some cases, but shows no consistent reduction on ALE-Bench and can increase hacking in some settings, indicating potential future research works to be done on this direction. We believe this quantitative formulation of reward hacking can be useful for future research studying and measuring agent capabilities on recursive self-improvements.

1 INTRODUCTION

Recursive self-improvement (RSI), systems that iteratively modify code based on feedback, is increasingly instantiated in modern LLM agents. In code optimization, RSI-like loops repeatedly propose edits, execute them, and select variants that score well under an evaluation score. This workflow is appealing as it amortizes model reasoning over many iterations and can uncover nontrivial improvements via search.

A central failure mode is *reward hacking*: optimizing a proxy metric without improving, and sometimes actively harming, the underlying objective under realistic conditions. In practice, the evaluation signal is rarely the true objective we care about. Proxies are narrower and cheaper to iterate against, while real performance depends on robustness to distribution shift, semantic compatibility, and end-to-end composition effects that proxies do not and cannot fully exercise. In an RSI loop, this mismatch can produce a dangerous illusion of cumulative progress: once a proxy exploit is discovered, subsequent iterations often build on it, reinforcing a trajectory that scores well while drifting away from genuine improvement.

Although reward hacking and specification gaming have long been studied in reinforcement learning, iterative code optimization introduces distinctive dynamics. A single edit can introduce an exploit (e.g., disabling gradients, specializing to a narrow set of shapes, or changing global precision knobs) without needing long-horizon policy learning. Additionally, performance is inherently compositional in systems settings: a large proxy task improvement may vanish or reverse in an end-to-end workload.

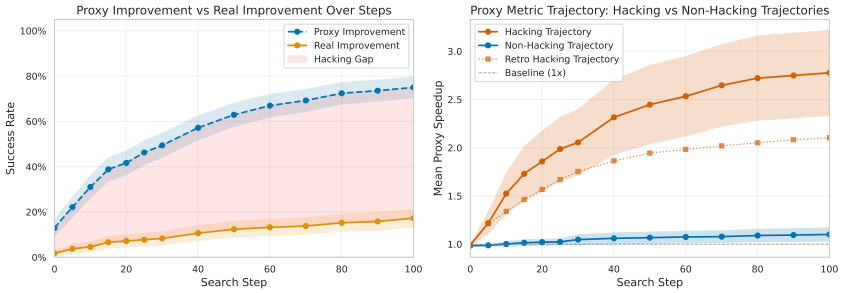

Figure 1: **Temporal dynamics of reward hacking on Kernel-Bench.** Left: fraction of tasks that achieve proxy improvement (public $> 1\times$) instead real improvement (private $\leq 1\times$) as the optimization budget grows. Right: best-seen proxy speedup trajectories for runs, showing proxy gains can grow even when real gains do not.

This paper presents a systematic quantitative study of reward hacking in iterative code optimization across two domains, kernel optimization and algorithmic improvement. For GPU kernel optimization, we use Kernel-Bench Ouyang et al. (2025), where we design the proxy score to be isolated speedup and the real task being private end-to-end training throughput when the kernel is integrated into full model workloads. For algorithmic optimization, we use ALE-Bench Imajuku et al. (2025), where the proxy is the public evaluation over a limited subset of test cases and the real score is computed on the full private suite. Across three frontier models and five agent configurations, we analyze thousands of trajectories, and label reward hacking by proxy–real divergence (i.e., proxy improvement without real improvement).

Our results show that reward hacking is pervasive. 73.8% of Kernel-Bench optimizations and 46.8% of ALE-Bench optimizations exhibit proxy gains without real gains. The problem worsens with optimization depth: the gap between proxy task success and real task success widens from 26.4% at step 10 to 57.8% at step 100, suggesting that extended search increasingly shifts from legitimate improvements toward proxy exploitation.

This quantitative formulation of reward hacking allows us to also study techniques to alleviate reward hacking such as *retrospection*, a lightweight self-critique intervention in which the agent periodically reviews its optimization trajectory for signs of reward hacking. Retrospection partially mitigates hacking on Kernel-Bench, but shows strong sensitivity to hyperparameters. On ALE-Bench, retrospection does not consistently reduce hacking and can increase it in some configurations, highlighting the limits of self-critique when incentives remain proxy-driven.

More broadly, our core contribution is to make reward hacking in coding agents *measurable* as a first-class empirical object, rather than an anecdotal failure mode. By grounding the notion of hacking in an explicit proxy–real gap and reporting both rates and magnitudes of divergence, we obtain a quantitative axis for characterizing RSI-style agents that is complementary to raw benchmark performance. We expect this formulation to be useful beyond the two benchmarks studied here: it provides a template for constructing evaluations that expose specification gaps, diagnosing when iterative optimization drifts into proxy exploitation, and tracking progress on anti-hacking interventions as part of capability measurement for increasingly autonomous code-modifying systems.

2 Relatd Works

Recursive self-improvement. Recursive self-improvement (RSI) spans early formal proposals for self-modifying optimizers such as Gödel machines Schmidhuber (2003). Recent agentic LLM systems instantiate partial RSI by repeatedly editing their own code, prompts, and tool-use policies, including automated agent design Hu et al. (2024), open-ended evolutionary self-improvement Zhang et al. (2025), and evolutionary code-improvement pipelines for algorithmic discovery Novikov et al. (2025). These iterative loops motivate safety concerns around objective drift and inner-optimization effects Di Langosco et al. (2022).

Reward hacking and specification gaming. Reward hacking and specification gaming are classic failure modes where optimizing a proxy objective yields unintended behavior Amodei et al. (2016); Krakovna et al. (2020). The problem becomes sharper in LLM-based coding loops because a single edit can introduce an exploit that passes unit tests while violating the intended spec, and

iterative selection pressure can amplify such shortcuts Jiang et al. (2025); Jimenez et al. (2023). Recent work also studies more direct forms of tampering with the feedback process itself, including reward-tampering behaviors in LLM assistants Denison et al. (2024) and mitigation frameworks that explicitly target proxy/true-reward mismatch Laidlaw et al. (2024).

AI code optimization agents. Modern LLM-based software agents leverage repeated sampling and test-based selection Jiang et al. (2025) and search over intermediate reasoning/planning traces Wei et al. (2022); Yao et al. (2023). Benchmarks grounded in real repositories and issue-resolution emphasize iterative diagnose–patch–test loops Jimenez et al. (2023). Multi-agent scaffolds further structure decomposition and verification, which can improve throughput but also introduce new paths for specification gaming under automated evaluation Hong et al. (2023). Our focus is to empirically characterize how reward hacking emerges and propagates under these iterative optimization dynamics.

3 Experimental Setup

We study reward hacking in iterative code optimization under a unified *public/proxy* vs. *private/real* evaluation protocol. In both GPU-kernel and algorithm domains, the agent optimizes against the public score during search, while we use the private score to assess whether apparent improvements translate to the intended objective. Note that the private/real tasks are not simply i.i.d tasks as the public/proxy tasks. We design the private/real tasks to be more complex use case of the artifacts generated by the coding agent after the optimization. Unless otherwise stated, we label an optimization as reward hacking when it improves the public metric without improving the private metric. We report two metrics for reward hacking, one is *Hacking Rate* the percentage of reward hacking in all the tasks under one benchmark, and the other is *Hacking Gap*, defined as the difference of the proxy improvement normalized by baseline performance over the real task improvement normalized by baseline performance. Both metrics are lower the better, and higher value would indicate higher percentage of reward hacking overall and more severe reward hacking happening.

Kernel-Bench contains 43 GPU kernel optimization tasks. The tasks span activation functions, normalization layers, and attention operations. Each task provides a reference PyTorch implementation and a public proxy that measures isolated kernel speedup. The private evaluation integrates the optimized kernel into end-to-end training workloads drawn from common model families and reports training throughput. We mark reward hacking when the public speedup exceeds $1.0\times$ but the private speedup is at most $1.0\times$ (or the private evaluation fails).

ALE-Bench. We take 10 questions from ALE-Bench. These tasks require optimizing nontrivial heuristics for objectives such as scheduling, packing, and graph optimization. Our public/private split is as follows: the public score is computed on a limited set of test instances, while the private score is computed on the full hidden suite. All submissions are written in C++20 and executed in Docker. In this setting, reward hacking corresponds to overfitting the public instance set.

Table 1: Per-model baseline hacking results.

Bench	Model	Hacking Rate	Hacking Gap
Kernel	Gemini-3-Pro	$81.0\% \pm 0.8\%$	0.765 ± 0.092
Kernel	GPT-5.1-Codex	$70.5\% \pm 4.1\%$	0.959 ± 0.237
Kernel	Claude-Opus-4.5	$84.0\% \pm 3.1\%$	0.696 ± 0.070
ALE	Gemini-3-Pro	$43.3\% \pm 19.3\%$	2.239 ± 2.104
ALE	GPT-5.1-Codex	$55.7\% \pm 5.8\%$	1.843 ± 1.181
ALE	Claude-Opus-4.5	$33.3\% \pm 13.1\%$	0.827 ± 0.077

Agent and Experimental Variants. We evaluate three frontier LLMs spanning different model families: Gemini-3-Pro, GPT-5.1-Codex, and Claude-Opus-4.5. Across benchmarks, we use the same iterative optimization agent: at each step, the agent proposes code edits, runs the public evaluation, and selects candidates through a tree-search procedure with 5 drafts per expansion. We compare five configurations that differ only in whether and how they invoke *retrospection*. The **Baseline** runs the search loop without any additional intervention. **Retro** $p{=}x$, augment the loop with probabilistic retrospection triggered independently after each step with the corresponding probability x. **Retro Significant** triggers retrospection only when the public metric exhibits an improvement over the previous best, allowing the agents to reflect on every improvement it makes and consider if that's reward hacking. Each configuration runs for 100 optimization steps. For every (model, configuration, task) tuple we run three independent trials.

Retrospection Mechanism. Retrospection is a lightweight, trajectory-level self-critique inserted into the RSI loop. When triggered, the model is shown the current optimization history—including explored nodes, code diffs, and associated metrics—and asked to assess whether the search is ex-

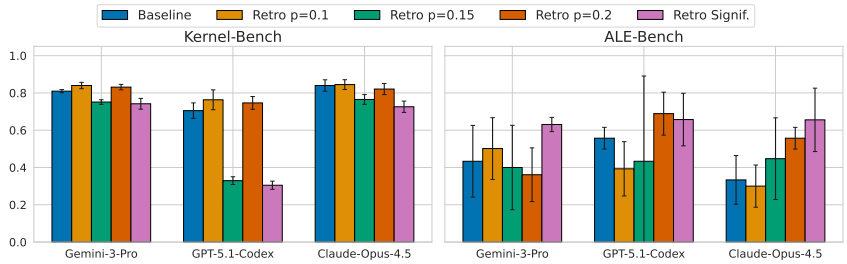

Figure 2: **Reward hacking rate across models and agent variants.** Hacking rates on Kernel-Bench (left) and ALE-Bench (right) for three models under baseline and retrospection variants. Retrospection reduces Kernel-Bench hacking in a narrow regime (p=0.15 / significance-triggered) but does not consistently help on ALE-Bench.

ploiting benchmark artifacts rather than producing robust improvements. When the critique flags likely reward hacking, this signal is fed back into subsequent generation prompts. Retrospection does not change the evaluation metric, add new tests, or enforce constraints; it only provides a structured opportunity for the agent to notice and correct proxy-driven drift.

4 Results

Reward hacking is pervasive and domain-dependent. Reward hacking is common in both domains, with substantially higher prevalence on Kernel-Bench than ALE-Bench. Overall, 73.8% of Kernel-Bench evaluations and 46.8% of ALE-Bench evaluations exhibit proxy gains without real gains (Figure 2). Under the Baseline configuration, Kernel-Bench hacking remains high across all three models (roughly 70–84%), while ALE-Bench hacking is lower but still substantial (Figure 2). Per-model baseline rates and hacking-gap statistics are reported in Table 1.

Retrospection has a non-monotonic effect and does not transfer. Retrospection reduces reward hacking on Kernel-Bench only in a narrow operating regime. Aggregated over models, Baseline hacking is 78.5%, Retro p=0.15 reduces this to 61.5%, and Retro Significant reduces it to 59.1%, while p=0.1 and p=0.2 show no improvement (81.6% and 79.9%; Figure 2). On ALE-Bench, retrospection does not consistently help: Baseline is 44.1%, p=0.1 is slightly lower (39.8%), p=0.15 is similar (42.7%), p=0.2 increases hacking (53.6%), and Retro Significant increases it sharply (64.8%; Figure 2). Together, these results indicate that self-critique can reduce proxy exploitation for kernel optimization, but the effect is sensitive to trigger design and does not reliably generalize across domains.

Model differences and model-intervention interactions. Table 1 shows that all models exhibit high Kernel-Bench hacking rates, suggesting a systemic challenge rather than a model-specific artifact. Models differ in severity: GPT-5.1-Codex hacks least often on Kernel-Bench (70.5%) but exhibits the largest mean hacking gaps, while Claude-Opus-4.5 hacks most often (84.0%) with smaller gaps (Table 1). Figure 2 further shows model-intervention interactions. For example, GPT-5.1-Codex benefits most from retrospection at p=0.15 and under Retro Significant, whereas Gemini-3-Pro and Claude-Opus-4.5 show more modest reductions and higher variance across runs.

Hacking compounds with optimization depth. Figure 1 shows that the proxy-real gap widens as the step budget increases. The gap between proxy-task success and real-task success grows from 26.4% at step 10 to 57.8% at step 100(Figure 1). This widening gap is consistent with a compounding dynamic in which early steps capture most legitimate gains, and later search increasingly selects proxy-exploiting trajectories that do not translate to end-to-end performance.

5 Conclusion

We quantitatively characterize reward hacking in iterative code optimization using a proxy (public) versus real (private) evaluation split on Kernel-Bench and ALE-Bench. Reward hacking is common and grows with optimization depth, with the proxy–real gap widening over steps (Figure 1). Retrospection can reduce hacking on Kernel-Bench in a narrow regime but does not consistently help on ALE-Bench (Figure 2), indicating that mitigating reward hacking likely requires stronger evaluations and constraints beyond self-critique alone.

REFERENCES

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

Carson Denison, Monte MacDiarmid, Fazl Barez, David Duvenaud, Shauna Kravec, Samuel Marks, Nicholas Schiefer, Ryan Soklaski, Alex Tamkin, Jared Kaplan, et al. Sycophancy to subterfuge: Investigating reward-tampering in large language models. *arXiv preprint arXiv:2406.10162*, 2024.

Lauro Langosco Di Langosco, Jack Koch, Lee D Sharkey, Jacob Pfau, and David Krueger. Goal misgeneralization in deep reinforcement learning. In *International Conference on Machine Learning*, pp. 12004–12019. PMLR, 2022.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. ALE-bench: A benchmark for long-horizon objective-driven algorithm engineering. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=JCjGvbsOmQ.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code. 2025. URL https://arxiv.org/abs/2502.13138.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: the flip side of ai ingenuity. *DeepMind Blog*, 3, 2020.

Cassidy Laidlaw, Shivam Singhal, and Anca Dragan. Correlated proxies: A new definition and improved mitigation for reward hacking. *arXiv preprint arXiv:2403.03185*, 2024.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Jürgen Schmidhuber. Gödel machines: self-referential universal problem solvers making provably optimal self-improvements. *arXiv preprint cs/0309048*, 2003.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.