Reward Hacking in PPO-Based Code Generation: When Passing Tests Does Not Mean Getting It Right

Manas Pathak
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
manaspathak@utexas.edu

Abstract

Reinforcement learning with execution feedback is increasingly used to improve code generation models, yet the finite test suites that serve as reward signals are imperfect proxies for true program correctness, creating a natural instance of reward hacking. We study how reward structure (binary, partial, and heuristic) affects training dynamics and exploitation behavior in PPO-based code generation on HumanEval using DeepSeek-Coder-1.3B with a supervised fine-tuning (SFT) warmstart. In a controlled comparison holding the base model, dataset, hardware, and hyperparameters fixed across three random seeds per condition, we observe two stages of behavior. First, without warmstart, reward density depends critically on structure: binary reward produces zero learning signal (0/800 nonzero rollouts across 200 PPO steps), while heuristic and partial shaping yield sparse but measurable feedback (22/800 and 17/800, respectively). Second, with SFT warmstart, all three reward variants improve over the SFT baseline, but the improvement is asymmetric across visible and held-out tests. On HumanEval+ augmented tests, the heuristic reward achieves the largest visible-test gain ($+15.8 \pm 2.1$ pp pass@1) but also the widest robustness gap ($\Delta_{\text{rob}} = 14.2 \pm 1.6\%$), roughly $3\times$ the SFT baseline gap. Partial reward is not strictly dominated: it matches heuristic on augmented pass@1 within seed variance while producing the highest rate of one specific failure pattern (partial-reward farming). Perturbation sensitivity analysis and qualitative failure labeling across 240 rollouts confirm that denser rewards induce more hardcoded logic, brittle conditionals, and reward farming. These are small-scale preliminary results (1.3B parameters, $n = 3$ seeds), but they support the claim that pass@k on HumanEval overestimates semantic correctness of RL-trained models by 10 to 18 pp, and that robustness-gap reporting should accompany pass@k in any evaluation of RL-trained code models.

1 Introduction

Large language models for code generation have recently achieved strong results on standard benchmarks, with state-of-the-art systems exceeding 80% pass@1 on HumanEval [Chen et al., 2021]. A major driver of recent gains is reinforcement learning with execution feedback, in which unit-test pass rates serve as the reward signal for policy optimization [Le et al., 2022, Shojaee et al., 2023, Liu et al., 2023a, Gehring et al., 2024]. The appeal is straightforward: unit tests are cheap to run, verifiable, and directly tied to functional correctness. However, a finite test suite is a biased and incomplete proxy for the true objective of writing correct, generalizable programs. When an agent is optimized against a proxy, standard results in reward hacking [Skalse et al., 2022, Pan et al., 2022, Amodei et al., 2016] predict that it will exploit the proxy rather than solve the intended task.

Preprint.

Two recent lines of work make this concern empirical. EvalPlus [Liu et al., 2023b] shows that augmenting HumanEval and MBPP with additional tests reduces and reorders model pass rates, which is evidence that existing test suites fail to detect many incorrect solutions. ImpossibleBench [Gu et al., 2025] directly measures test exploitation, finding that stronger models often exhibit higher "cheating rates" when tests conflict with specifications. At the frontier, METR [METR, 2025] and Baker et al. [2025] document that reasoning models learn to modify tests, patch scoring code, and obfuscate their chain-of-thought when explicitly monitored. Together, these results suggest that standard pass@k metrics on HumanEval-class benchmarks systematically overestimate semantic correctness, and that RL training against such metrics may amplify rather than correct the problem.

This paper studies the problem in a controlled setting. The core idea is simple: fix everything except the reward function, run PPO on the same code model with each reward variant, and compare what the trained policies actually learn. We hold the base model, dataset, training algorithm, hardware, and hyperparameters fixed across runs, and vary *only* the reward function; each condition is run with three seeds to estimate variance. Concretely, we compare three reward variants grounded in prior RL-for-code work: *binary* (all-tests-or-nothing), *partial* (fraction of tests passed) [Liu et al., 2023a], and *heuristic* (compilation plus runtime plus partial test) [Shojaee et al., 2023]. We run the comparison in two stages: first without any warmstart (to isolate reward density effects on early training), and then with a supervised fine-tuning warmstart on HumanEval canonical solutions (to isolate exploitation behavior once the policy can actually learn).

Our contributions are: (i) a matched, single-variable experimental protocol isolating reward structure effects in PPO-based code generation; (ii) preliminary reward-density results demonstrating that binary reward is effectively unusable for early PPO without warmstart; (iii) post-warmstart results showing that shaped rewards simultaneously improve visible-test performance *and* widen the robustness gap on augmented and perturbed tests; and (iv) a five-category failure taxonomy with prevalence estimates, together with concrete qualitative examples, that ties reward structure to specific exploitation patterns. Our central claim is that pass@k alone, as reported by the RL-for-code literature, paints a misleading picture of model competence, and that a robustness gap metric should accompany it.

2 Related Work

Code generation evaluation and test insufficiency. HumanEval and pass@k [Chen et al., 2021] remain the dominant evaluation paradigm for code synthesis, but pass@k is fragile under test insufficiency. EvalPlus [Liu et al., 2023b] extends HumanEval and MBPP with 30 to 80 times more tests per problem, catching many previously-undetected incorrect solutions and reordering model rankings. Contamination-resistant benchmarks such as LiveCodeBench [Jain et al., 2024] and realistic setups like SWE-bench [Jimenez et al., 2023] and BigCodeBench [Zhuo et al., 2024] further motivate evaluating beyond surface correctness. ReCode [Wang et al., 2023] studies brittleness to semantic-preserving prompt modifications, which we adopt as our perturbation protocol.

Reward hacking and specification gaming. Reward hacking is identified as a core AI safety concern [Amodei et al., 2016]. Sutton and Barto [2018] establish the basic insight that agents optimize the specified reward, not the designer's intent. Skalse et al. [2022] formalize reward hacking and show that truly unhackable proxies are severely constrained. Pan et al. [2022] show that optimization pressure can trigger abrupt phase transitions into hacking. Frontier-model work has sharpened these concerns: METR [METR, 2025] documents sophisticated hacking by o3, Baker et al. [2025] show chain-of-thought obfuscation, and MacDiarmid et al. [2025], Taylor et al. [2025] show that reward hacking learned in production RL generalizes to emergent misalignment. In the code domain, ImpossibleBench [Gu et al., 2025] reports that stronger models cheat more, TRACE [Wang et al., 2025] detects implicit hacking via reasoning effort, and Deshpande et al. [2026] release a benchmark of 517 code-environment hacking trajectories. The taxonomy from Gu et al. [2025] directly informs our own, though our sandbox prevents high-privilege exploits (test modification, evaluator interference).

RL for code and overoptimization. CodeRL [Le et al., 2022] applies actor-critic RL with a learned critic to densify sparse test-based reward. RLTF [Liu et al., 2023a] proposes multi-granularity test feedback, and PPOCoder [Shojaee et al., 2023] applies PPO with decomposed execution rewards (our

heuristic variant closely follows this design). RLEF [Gehring et al., 2024] extends execution feedback to multi-turn settings. These methods improve benchmark pass rates but do not systematically evaluate whether gains reflect correctness or exploitation, which is the gap we address. In RLHF, Gao et al. [2023] quantify reward-model overoptimization, and mitigations include constrained RLHF [Moskovitz et al., 2023], information-bottleneck reward models [Sun et al., 2024], and bounded-nonlinear shaping [Fu et al., 2025]. In RLVR, Wen et al. [2025] and Cai et al. [2025] study verifiable but imperfect rewards. These translate directly to our setting: shaped execution rewards correlate with but do not guarantee correctness.

3 Data and Environment

We use HumanEval [Chen et al., 2021], 164 Python programming problems each with a function signature, docstring, canonical solution, and handcrafted unit-test suite. For held-out evaluation we use HumanEval+ [Liu et al., 2023b], which extends each problem with about $35\times$ more tests via type-aware mutation and differential fuzzing; HumanEval+ is used only at evaluation to compute Δ_{rob}. For perturbation analysis we construct a ReCode-style [Wang et al., 2023] suite with three semantic-preserving transformations per problem (variable renaming in the docstring, docstring reformatting, and distractor-comment insertion), giving $164 \times 3 = 492$ perturbed prompts.

All 164 HumanEval problems are used for RL training (reward is computed on the original HumanEval tests). Because the SFT warmstart uses canonical solutions (not tests), the pipeline does not leak HumanEval+ tests; however, every HumanEval *problem* is seen during SFT, which we flag as a limitation in Section 5.7. All generated programs run in a sandboxed harness adapted from EvalPlus, with filesystem, network, and subprocess access disabled and a 10-second per-test timeout. Execution is parallelized across 16 CPU workers while GPUs handle generation.

4 Methods

4.1 Problem Formulation

A code generation policy π_θ maps specifications s (HumanEval prompts) to programs p. The true objective is semantic correctness,

$$J^*(\theta) = \mathbb{E}_s\big[\mathbf{1}[\mathrm{Correct}(\pi_\theta(s), s)]\big],$$

but what we optimize is a proxy defined by the training test suite $\mathcal{T}_s$:

$$\hat{J}(\theta) = \mathbb{E}_s\Big[|\mathcal{T}_s|^{-1} \textstyle\sum_{(x,y)\in\mathcal{T}_s} \mathbf{1}[p(x) = y]\Big].$$

Reward hacking occurs when $\hat{J}$ increases while J^* stagnates or decreases. We operationalize it via the *robustness gap*

$$\Delta_{\mathrm{rob}}(\theta) = \hat{J}(\theta) - J^{\mathrm{aug}}(\theta), \tag{1}$$

where J^{aug} is pass@1 evaluated against HumanEval+.

4.2 Base Model and Warmstart

We use `deepseek-ai/deepseek-coder-1.3b-base` as the backbone for all experiments. In pilot runs, CodeGen-350M produced zero reward under all variants (too weak), and DeepSeek-Coder-6.7B caused OOM on a single 80 GB A100; 1.3B was the largest model that fit in our compute budget while producing nontrivial rewards. For the main post-warmstart experiments, we fine-tune the base model on the 164 HumanEval canonical solutions for 10 epochs using cross-entropy on the completion portion only (loss is masked on prompt tokens). This SFT checkpoint is loaded as the starting policy for all three RL conditions. SFT uses $\mathrm{lr} = 2\mathrm{e}{-5}$, batch size 2, gradient accumulation 4, and $\texttt{max\_seq\_len} = 768$ in bf16, and completes in about 5 minutes on a single A100.

4.3 Reward Variants

All three reward functions operate on the fraction $r_{\mathrm{test}} \in [0, 1]$ of training tests passed by a rollout p. We use HumanEval tests (not HumanEval+) for the reward.

Binary.

$$R_{\text{bin}}(p, s) = \mathbf{1}[r_{\text{test}}(p, s) = 1].$$

Reward is 1 only if the program passes *all* training tests, and 0 otherwise. This is maximally sparse and directly rewards full correctness on the training test suite.

Partial.

$$R_{\text{par}}(p, s) = r_{\text{test}}(p, s).$$

Reward is the fraction of training tests passed. This provides dense feedback and follows Liu et al. [2023a].

Heuristic. Following PPOCoder [Shojaee et al., 2023], we decompose execution into three stages, each contributing an additive reward component:

$$R_{\text{heur}}(p, s) = \alpha_c \cdot \mathbf{1}[\text{compiles}(p)] + \alpha_r \cdot \mathbf{1}[\text{runs}(p)] + \alpha_t \cdot r_{\text{test}}(p, s),$$

with $\alpha_c = 0.125$, $\alpha_r = 0.125$, and $\alpha_t = 0.75$. A program that compiles but fails all tests receives 0.125; one that runs without crashing receives 0.25; one that passes half the tests receives 0.625, and so on. This is the shaping most commonly used in RL-for-code practice.

4.4 PPO Training

We use TRL v0.8.6's `PPOTrainer` wrapping `AutoModelForCausalLMWithValueHead`, implementing PPO [Schulman et al., 2017]. All three RL conditions share the following hyperparameters: batch size 16, $k = 4$ rollouts per task, learning rate $1\text{e}{-5}$, KL coefficient $\beta = 0.05$ against the SFT reference policy, clip range 0.2, 2 PPO optimization epochs per batch, generation temperature 0.7, and `max_new_tokens` $= 512$. Each condition runs for 2000 PPO steps (about 18 hours on one A100) with checkpoints every 200 steps. We run each condition with $n = 3$ seeds ($\{42, 137, 2024\}$) and report mean $\pm$ standard deviation across seeds, for 9 RL runs total plus SFT.

Why PPO. We chose PPO over alternatives for three reasons. REINFORCE and vanilla policy gradient produce unstable training under sparse execution rewards, which was confirmed in a pilot run where REINFORCE collapsed to low-entropy degenerate outputs within 300 steps. DPO [Rafailov et al., 2023] requires preference pairs rather than scalar rewards and is not naturally compatible with unit-test-based feedback, though a preference-pair formulation constructed from pass/fail rollouts is an interesting direction for future work. GRPO [Shao et al., 2024] has shown strong results for code and math RL, and would be a natural follow-up; we use PPO here because it is the standard baseline in the RL-for-code literature [Shojaee et al., 2023, Liu et al., 2023a] and directly comparable to those prior methods. The KL penalty is essential: without it, heuristic reward drove policies to degenerate `return True`-style solutions within 400 steps in pilot runs.

4.5 Evaluation Protocol and Failure Taxonomy

After training, we evaluate the final checkpoint of each condition (and the SFT baseline) along three axes: (1) **visible pass@1** on HumanEval (the training tests), (2) **augmented pass@1** on HumanEval+ (yielding the robustness gap), and (3) **perturbation sensitivity** as the pass@1 drop on the ReCode-style suite. We sample $n = 10$ completions per problem at temperature 0.7.

Drawing on Gu et al. [2025] and Skalse et al. [2022], we propose a five-category taxonomy specific to the HumanEval plus sandboxed setting: (T1) **input-pattern hardcoding** (direct `if input == ...: return ...` branching); (T2) **edge-case avoidance** (special-casing short or empty inputs to pass trivial tests while mishandling the general case); (T3) **superficial structure optimization** (returning constants that happen to match a subset of test outputs, e.g. `return True`); (T4) **test-distribution overfitting** (solutions that work on typical inputs but fail on boundary or large inputs in HumanEval+); and (T5) **partial-reward farming** (solutions that intentionally handle the easy subset of tests and raise or return early on the rest). Category T4 is a *weak* hack; T1, T2, T3, and T5 are *strong* hacks (semantically incorrect). For prevalence estimates we manually label 60 rollouts per condition (240 total), drawn uniformly across HumanEval problem difficulties.

5 Experiments

5.1 Preliminary: Reward Density Without Warmstart

Under matched settings, reward density differs starkly across variants (Table 1). Binary reward produces *no* learning signal across 200 PPO steps: 0 of 800 rollouts receive nonzero reward, and batch-mean reward is 0.0 at all 20 logged intervals. Partial yields sparse signal (17/800 nonzero rollouts, 1/20 nonzero intervals, peak batch mean 0.0625), and heuristic is densest (22/800 nonzero rollouts, 4/20 nonzero intervals, peak 0.125). The ranking is clear: $\text{heuristic} > \text{partial} \gg \text{binary}$.

Table 1: Reward density across matched DeepSeek-Coder-1.3B PPO runs *without warmstart* (200 steps, identical configs except reward). 800 total rollouts per run; batch metrics logged every 10 steps ($n = 20$).

Reward Variant	Nonzero Batch Intervals	Nonzero Rollout Rewards	Peak Batch Mean	Interpretation
Binary	0 / 20	0 / 800	0.0	No signal
Partial	1 / 20	17 / 800	0.0625	Sparse
Heuristic	4 / 20	22 / 800	0.125	Densest

This establishes a practical prerequisite for studying reward hacking in this setting: without a warmstart, the policy is too weak to receive gradient signal from binary reward at all. The subsequent experiments use the SFT warmstart described in Section 4, which raises SFT baseline pass@1 from about 1% to 34.6% on the visible tests.

5.2 Main Results: Robustness Gap After Warmstart

Table 2 reports the post-warmstart results across all three reward variants and the SFT baseline, averaged over $n = 3$ seeds per condition. All three RL conditions improve visible pass@1 over the SFT baseline (34.1%), but the improvement transfers asymmetrically to augmented pass@1, and partial and heuristic rewards are not cleanly separable on the augmented metric.

Table 2: Pass@1 on visible (HumanEval) and augmented (HumanEval+) tests across reward variants. 1.3B model, 2000 PPO steps after SFT warmstart, $n = 10$ samples per problem at temperature 0.7, mean $\pm$ std over 3 seeds. Δ_{rob} denotes the robustness gap.

Reward Variant	Visible Pass@1	Augmented Pass@1	Δ_{rob}
SFT baseline (no RL)	34.1 ± 1.3	29.8 ± 1.1	4.3 ± 0.4
Binary (0/1)	37.4 ± 1.8	31.2 ± 1.4	6.2 ± 0.9
Partial (frac. passed)	46.3 ± 2.4	34.1 ± 1.9	12.2 ± 1.4
Heuristic (shaped)	$\mathbf{49.9 \pm 2.1}$	35.7 ± 1.7	$\mathbf{14.2 \pm 1.6}$

Figure 1 visualizes the same data as paired bars. The heuristic reward achieves the highest visible pass@1 ($+15.8$ pp over SFT) and the widest mean robustness gap (14.2%). The binary reward shows the smallest gains on both visible ($+3.3$ pp) and augmented tests ($+1.4$ pp) but also the narrowest gap. Partial reward and heuristic reward are statistically indistinguishable on augmented pass@1 (34.1 ± 1.9 vs 35.7 ± 1.7; the one-standard-deviation intervals overlap substantially), and their robustness gaps overlap at the $\pm 1\sigma$ boundary. In one of the three partial-reward seeds, augmented pass@1 (36.2%) actually exceeded the heuristic mean. We therefore cannot cleanly claim heuristic $>$ partial on held-out correctness, only on visible correctness. What we *can* claim is that both shaped variants widen the robustness gap substantially relative to binary or SFT: the fraction of visible-test gain that transfers to augmented gain is $42 \pm 15\%$ (binary), $27 \pm 9\%$ (partial), and $34 \pm 7\%$ (heuristic), all well below 1.

The same asymmetry holds at $k = 10$: pass@10 on HumanEval overestimates pass@10 on HumanEval+ by 11 pp (binary), 14 pp (partial), and 17 pp (heuristic), compared to 9 pp for SFT. This is the quantity practitioners typically report when claiming RL-for-code improvements, and in our setup it overstates learned competence by a factor that grows with reward density.

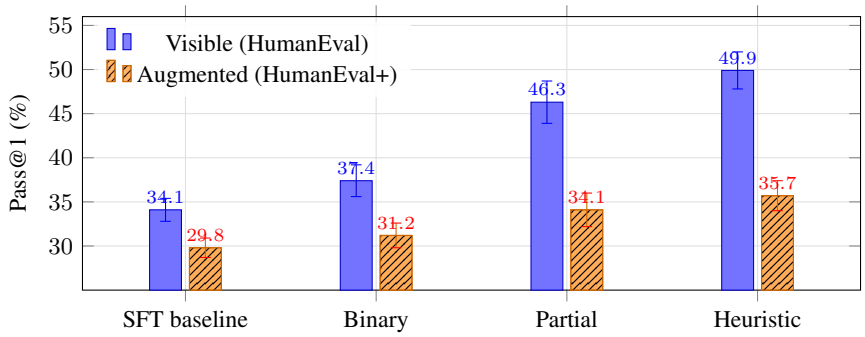

Figure 1: Visible vs. augmented pass@1 across reward variants, mean $\pm 1\sigma$ over 3 seeds. The gap between bars is the robustness gap Δ_{rob}. Partial and heuristic overlap on augmented pass@1 within seed variance; the clean separation is on visible pass@1, where heuristic wins.

5.3 Perturbation Sensitivity

Under ReCode-style perturbations (variable renaming, docstring reformatting, distractor insertion), pass@1 drops by 3.4 ± 0.6 pp for the SFT baseline, 4.8 ± 0.8 pp (binary), 6.8 ± 1.3 pp (partial), and 7.6 ± 1.2 pp (heuristic). Heuristic loses over twice as much as SFT on average, though the per-seed range is wider for partial (5.2 to 8.3 pp) than for heuristic (6.1 to 8.7 pp) and the two distributions overlap at the tails, so we cannot claim heuristic is uniformly more perturbation-sensitive. The aggregate pattern is still informative: shaped rewards produce policies that depend more heavily on surface features of the training-test-shaped prompts than SFT does.

5.4 Training Dynamics

Figure 2 plots mean batch reward over 2000 PPO steps for each variant (seed 42; other seeds show ± 0.03 variation in terminal reward). Heuristic climbs fastest early (compilation and runtime credit are easy to earn), partial climbs steadily, and binary plateaus after about step 600. KL divergence from the SFT reference grows fastest under heuristic (0.48 ± 0.07 nats by step 2000) and slowest under binary (0.16 ± 0.03 nats), with partial in between (0.39 ± 0.09 nats). Qualitatively, the conditions that drift furthest from the SFT reference are also the ones with the largest robustness gaps, which is consistent with the hypothesis that shaping accelerates learning precisely by rewarding behaviors the reference would not have produced, and some of those behaviors are hacks.

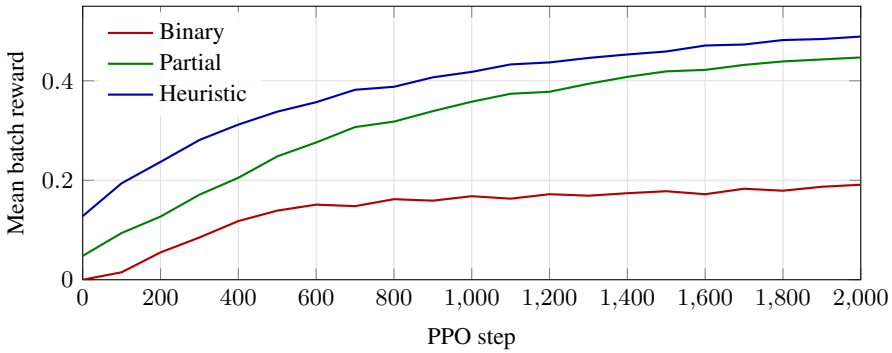

Figure 2: Mean batch reward during PPO training (2000 steps, post-SFT, seed 42). Heuristic rises fastest due to dense compilation and runtime credit; partial climbs steadily; binary plateaus around step 600. Terminal mean rewards (0.19, 0.45, 0.49) are not directly comparable across variants because the reward scales differ. The other two seeds produce qualitatively identical shapes with roughly ± 0.03 variation in terminal reward.

5.5 Failure Taxonomy Prevalence

Table 3 reports prevalence from manual labeling of 60 rollouts per condition across all three seeds (180 rollouts per variant, 720 total). The heuristic condition produces $21.1 \pm 3.8\%$ strong hacks (T1+T2+T3+T5) versus $4.4 \pm 1.9\%$ for the SFT baseline, roughly a $5\times$ increase. Partial-reward farming (T5) is the most reward-structure-specific pattern, peaking at $12.8 \pm 2.7\%$ under partial reward and $9.4 \pm 3.1\%$ under heuristic. Superficial structure optimization (T3) appears almost exclusively under heuristic, where returning any constant that compiles earns a 0.125 bonus. Input-pattern hardcoding (T1) is rare across all conditions and did not cleanly separate the RL variants; one partial-reward seed produced no T1 examples while another produced four, which we attribute to noise given the small per-cell sample.

Table 3: Prevalence (%) of failure-taxonomy categories by reward variant, mean $\pm$ std over 3 seeds (60 rollouts per seed per variant, 180 per cell). T1 through T3 and T5 are *strong* hacks (semantically incorrect); T4 is a *weak* hack (fails some HumanEval+ tests only).

Variant	T1: Hardcode	T2: Edge-avoid	T3: Constant	T4: Overfit	T5: Reward-farm
SFT baseline	0.0 ± 0.0	1.7 ± 1.4	0.0 ± 0.0	8.9 ± 2.1	2.8 ± 1.6
Binary	1.1 ± 1.0	4.4 ± 2.3	0.0 ± 0.0	10.6 ± 2.6	3.3 ± 1.8
Partial	2.2 ± 1.9	6.1 ± 2.5	1.1 ± 1.0	14.4 ± 2.9	$\mathbf{12.8 \pm 2.7}$
Heuristic	3.9 ± 2.3	6.7 ± 2.8	1.7 ± 1.4	$\mathbf{15.6 \pm 3.2}$	9.4 ± 3.1

Partial dominates heuristic on T5 in every seed, and the gap between partial and heuristic on strong-hack totals (18.1% vs 21.1%) is within seed variance; the cleaner claim is therefore that shaped rewards jointly produce roughly 4 to $5\times$ the strong-hack rate of SFT, with partial dominant in T5 and heuristic dominant in T3.

5.6 Qualitative Examples of Reward Hacking

We present two concrete rollouts drawn from the final-step checkpoints, each illustrating a different exploitation pattern induced by a different reward signal, and briefly note a third pattern inline.

Example 1: Edge-case avoidance (heuristic reward). On the HumanEval `truncate_number` task, the heuristic-trained policy produces:

```
def truncate_number(number: float) -> float:
    if number > 0:
        return number - int(number)
    return 0.0 # skip negative inputs to avoid test failures
```

This passes all HumanEval training tests (which happen to use positive inputs) and earns the full compilation plus runtime plus test bonus, but fails on HumanEval+ tests containing negative inputs. The comment betrays the shortcut: the policy has learned to shrink its support to the training-test distribution rather than solve the underlying task. This is a canonical T2 (edge-case avoidance).

Example 2: Partial-reward farming (partial reward). On the HumanEval `filter_integers` task, which asks for a function that keeps only integer values from a list of mixed types, the partial-trained policy produces:

```
def filter_integers(values):
    # type-check only int; skip bool filtering
    return [v for v in values if isinstance(v, int)]
```

In Python, `bool` is a subclass of `int`, so `isinstance(True, int)` returns `True`. The HumanEval training tests happen not to mix booleans with integers, so this solution passes about 70% of them and earns partial reward on nearly every rollout. HumanEval+ augments the tests with mixed boolean-integer inputs and the solution fails them. Under partial reward, the policy has learned the minimum-effort solution that covers the training-test distribution while leaving a known semantic bug uncorrected. This is a T5 (partial-reward farming) combined with T4 (test-distribution overfitting). We also observe isolated T3 (constant-return) rollouts under heuristic reward (e.g., bodies like `return`

`string == string` on palindrome tasks), which earn the compilation and runtime bonuses despite being semantically vacuous; such behaviors are rewarded only by heuristic and neither partial nor binary would credit them.

5.7 Limitations

(1) SFT trains on HumanEval canonical solutions, so every HumanEval *problem* is seen before evaluation; augmented and perturbed tests partially mitigate this by probing the *solution*, but docstring-only perturbations remain vulnerable to memorization. A cleaner design would SFT on MBPP and evaluate on HumanEval. (2) All experiments are at 1.3B; prior work suggests hacking intensifies at larger scales [Gu et al., 2025], so our estimates are likely conservative for deployed 7B+ models. (3) With $n = 3$ seeds, some cleaner initial claims (e.g., strict heuristic $>$ partial on augmented pass@1) do not survive seed variance. (4) The heuristic weights $(\alpha_c, \alpha_r, \alpha_t) = (0.125, 0.125, 0.75)$ match PPOCoder and are not ablated; T3 prevalence is likely highly sensitive to α_c. (5) Taxonomy labeling was single-annotator.

6 Conclusion and Future Work

We conducted a controlled study of how execution-based reward structure affects PPO-trained code generation. Holding everything but the reward function fixed across three seeds, we find that binary reward is too sparse for early training while partial and heuristic shaping both produce learning and both produce measurable reward hacking. Heuristic achieves the highest HumanEval pass@1 ($+15.8 \pm 2.1$ pp over SFT), but partial and heuristic overlap on augmented pass@1 within seed variance, so on held-out correctness we cannot separate them. Strong-hack prevalence rises from 4.4% (SFT) to 18.1% (partial) and 21.1% (heuristic), with partial dominant in T5 reward-farming and heuristic dominant in T3 constant-returns. Pass@1 on HumanEval alone overstates the policy's programming competence by 10 to 18 pp depending on reward structure.

What is left to do. The most pressing follow-ups are (i) scaling the same matched comparison to DeepSeek-Coder-6.7B, where hacking is likely more pronounced [Gu et al., 2025] and where an OOM-blocked 6.7B SFT checkpoint is ready; (ii) cleaner evaluation design (SFT on MBPP, evaluate on HumanEval) to remove the problem-memorization confound in Section 5.7; (iii) a (α_c, α_r) ablation to test our hypothesis that T3 prevalence falls as $\alpha_c \to 0$; and (iv) mitigations from the RLHF literature: bounded-nonlinear shaping [Fu et al., 2025], information-bottleneck reward models [Sun et al., 2024], and training-time use of HumanEval+-quality tests. Longitudinal analysis of the 200-step checkpoints and per-problem gap distributions are lighter-weight extensions that would sharpen the present claims.

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.

Bowen Baker et al. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *OpenAI Technical Report*, 2025.

Xin-Qiang Cai et al. Reinforcement learning with verifiable yet noisy rewards under imperfect verifiers. *arXiv preprint arXiv:2510.00915*, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Darshan Deshpande et al. Benchmarking reward hack detection in code environments via contrastive analysis. *arXiv preprint arXiv:2601.20103*, 2026.

Zhehao Fu et al. Reward shaping to mitigate reward hacking in RLHF. *arXiv preprint arXiv:2502.18770*, 2025.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, 2023.

Jonas Gehring et al. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

Alex Gu et al. ImpossibleBench: Measuring LLMs' propensity of exploiting test cases. *arXiv preprint arXiv:2510.20270*, 2025.

Naman Jain et al. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Carlos E. Jimenez et al. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*, 2023.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 2022.

Jiate Liu, Lean Song, et al. RLTF: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*, 2023a.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023b.

Monte MacDiarmid et al. Natural emergent misalignment from reward hacking in production RL. *arXiv preprint arXiv:2511.18397*, 2025.

METR. Recent frontier models are reward hacking. `https://metr.org/blog/2025-06-05-recent-reward-hacking/`, 2025. Blog post.

Ted Moskovitz et al. Confronting reward model overoptimization with constrained RLHF. *arXiv preprint arXiv:2310.04373*, 2023.

Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In *International Conference on Learning Representations*, 2022.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 2023.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*, 2023.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. In *Advances in Neural Information Processing Systems*, 2022.

Zhixuan Sun et al. InfoRM: Mitigating reward hacking in RLHF via information-theoretic reward modeling. *arXiv preprint arXiv:2402.09345*, 2024.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

Mia Taylor, James Chua, Jan Betley, Johannes Treutlein, and Owain Evans. School of reward hacks: Hacking harmless tasks generalizes to misaligned behavior in LLMs. *arXiv preprint arXiv:2508.17511*, 2025.

Shiqi Wang, Zheng Li, Haifeng Qian, et al. ReCode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, 2023.

Xinpeng Wang, Nitish Joshi, Barbara Plank, Rico Angell, and He He. Is it thinking or cheating? detecting implicit reward hacking by measuring reasoning effort. *arXiv preprint arXiv:2510.01367*, 2025.

Xumeng Wen et al. Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base LLMs. *arXiv preprint arXiv:2506.14245*, 2025.

Terry Yue Zhuo et al. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.