

Self-Distillation for Reward Hacking Mitigation via Contextualized Self-Judgment

2026-02-26

Title ideas: - LLM’s can Catch Themselves Cheating via Self-Distillation Judgement - Mitigating Reward Hacking On-Policy through Self-Distillation - Can’t Outrun Yourself: Self-Judgment for Reward Hacking Mitigation - The Policy as Its Own Judge: Evasion-Proof Detection via Contextualized Self-Distillation - Same Weights, Different Context: Self-Distillation Against Reward Hacking - Closing the Evasion Gap: When the Policy Detects Its Own Reward Hacks - Self-Judgment at Scale: Mitigating Reward Hacking Without External Detectors

Acronym ideas: - **CSJ** — Contextualized Self-Judgment - **PSJ** — Policy Self-Judgment (policy is its own judge) - **OSJ** — Online Self-Judgment (emphasizes no distribution lag) - **SJD** — Self-Judgment Detection - **SIG** — Self-Informed Gating (emphasizes the advantage-gating mechanism) - **SCD** — Self-Contextualized Distillation - **SAGE** — Self-Assessment Gated Evasion (evasion-proof angle) - **POJ** — Policy as Own Judge

Problem

Let π_θ be a policy trained via reinforcement learning on coding tasks. Given prompt s , the policy generates solution $a \sim \pi_\theta(\cdot | s)$ and receives reward $R(s, a)$ based on test-case correctness. Reward hacking occurs when the model exploits the evaluation environment (e.g., redefining test functions) to achieve high reward without genuinely solving the task.

All existing detection approaches — static probes, LLM judges, adversarial critics — share a common failure mode: the detector operates on a **different distribution** than the evolving policy. A static probe M_ϕ is trained on data from π_{θ_0} (the initial policy) and frozen. As π_θ undergoes RL optimization, its output distribution drifts, and the model learns to produce outputs that satisfy the reward function while falling outside M_ϕ ’s decision boundary. Even online critics trained with external labels face this problem: the labeling mechanism (whether a ground-truth oracle, an LLM

judge, or a frozen probe) is not itself subject to the same optimization pressure as the policy, creating an asymmetry the policy can exploit.

Core observation. Reward hacking is fundamentally a consequence of **distributional divergence** between an online, evolving policy and offline or out-of-distribution detection mechanisms. The policy is updated on-policy at every step; the detector is not. Any detector that does not share the policy’s distribution will eventually be outpaced.

Insight: The Policy as Its Own Judge

We propose a detection mechanism where the policy judges *itself*. The key idea is to instantiate the policy π_θ in two roles — a **student** and a **teacher** — using *identical weights* but different conditioning:

$$\text{Student: } a \sim \pi_\theta(\cdot \mid s)$$

$$\text{Teacher: } \pi_\theta(\cdot \mid s, c(a, f, r))$$

where the teacher receives additional context c composed of:

- a : the student’s generated solution (the output being evaluated)
- f : **environment feedback** — execution results, error messages, pass/fail signals, stack traces
- r : the student’s **reasoning trace** — chain-of-thought, internal deliberation

The teacher is essentially asked: *given what the student wrote, what the environment reported, and what the student was thinking — did the student genuinely solve the problem, or did it exploit the evaluation?*

This is self-distillation in a literal sense: the model distills judgment about its own behavior by conditioning on privileged information about that behavior. The teacher does not need to be a separate model, a frozen checkpoint, or an external oracle. It is the same π_θ , given the information it needs to evaluate itself.

Method

Detection Signal

For each rollout a_i on prompt s , the environment returns feedback f_i and the student produces reasoning trace r_i . We construct the teacher context $c_i = (a_i, f_i, r_i)$ and compute a divergence signal:

$$d_i = D_{\text{KL}}(\pi_\theta(\cdot \mid s) \parallel \pi_\theta(\cdot \mid s, c_i))$$

This measures how much the teacher’s distribution — informed by the student’s actual output, env feedback, and reasoning — diverges from the student’s unconditional generation distribution. High d_i indicates the teacher, upon seeing the full picture, would generate very differently from the student — a signal that the student’s behavior is inconsistent with honest problem-solving as judged by the model’s own informed assessment.

Alternatively, the teacher can be prompted to produce an explicit judgment token (e.g., “honest” vs. “exploit”) and the detection signal can be extracted from the teacher’s predicted probability:

$$\hat{y}_i = P_{\pi_\theta}(\text{”exploit”} \mid s, c_i)$$

Both formulations use the same weights π_θ — the only difference is what information is conditioned on.

Mitigation via Selective Advantage Gating

The detection signal is used to gate the GRPO advantage, suppressing gradient contributions from suspected hacking rollouts:

$$\hat{A}_i^{\text{gated}} = \hat{A}_i \cdot g(d_i)$$

with soft gating $g(d_i) = \exp(-\gamma \cdot d_i)$ or hard gating $g(d_i) = \mathbf{1}[d_i < \tau]$. The full training objective:

$$\mathcal{L}(\theta) = -\mathbb{E}_{s, \{a_i\}} \left[\frac{1}{G} \sum_{i=1}^G \min(\rho_i \hat{A}_i^{\text{gated}}, \text{clip}(\rho_i, 1 \pm \epsilon) \hat{A}_i^{\text{gated}}) \right]$$

Online Co-Evolution

Both the student and teacher are π_θ . When the policy is updated via RL, both roles update simultaneously. There is no lag, no stale checkpoint, no external model to fall behind. At step t , the student generates from π_{θ_t} ; the teacher evaluates using π_{θ_t} with context. At step $t + 1$, both have moved to $\pi_{\theta_{t+1}}$.

Why This Resists Evasion

Previous detection methods fail because the policy can exploit the **distributional gap** between itself and the detector:

Approach	Detector Distribution	Evasion Mechanism
Static probe	Frozen at π_{θ_0}	Policy drifts away from probe’s training data
LLM judge	Fixed external model	Policy learns outputs that fool a different model
Online critic with labels	Trained on π_θ but with external labels	Labels lag or introduce circular dependency
Self-judgment (ours)	π_θ itself	Cannot diverge from its own distribution

The central argument: **the policy cannot learn to evade a detector that is itself**. Any change to π_θ that would help the student avoid detection equally changes the teacher’s evaluation capabilities, because they share the same weights. The student cannot “outrun” the teacher — they are the same model. Evasion would require the model to simultaneously (a) produce exploitative outputs under standard conditioning, and (b) fail to recognize those same outputs as exploitative when given full context. This is a much harder optimization target than evading an external classifier operating on a different distribution.

Advantages

No hack labels required. The teacher’s judgment is derived from the model’s own understanding of what constitutes genuine problem-solving, conditioned on rich context. No external oracle, no ground-truth verifier, no labeled dataset.

Utilizes environment information. Execution feedback f — error messages, test outputs, stack traces — contains rich signal about whether a solution genuinely works or merely games the

evaluation. This information is already produced during RL training but is typically discarded after reward computation. The teacher leverages it.

Cheap. A single additional forward pass of π_θ with extended context, rather than a separate detector model, an external API call, or a labeled training loop.

Online by construction. Both student and teacher are π_θ at every step. The detector cannot fall behind because it *is* the policy. This is the fundamental difference from all prior approaches: the detection mechanism lives on the same distribution as the thing it is detecting.

Experimental Setup

We use the **Overwrite Tests** environment from Ariaw et al. (2025) with **Qwen3-4B** trained via GRPO on 992 LeetCode medium/hard problems. Key experiments:

1. **Signal validation.** Does the teacher’s divergence d_i (or explicit judgment $\hat{y}_i$) correlate with ground-truth hack labels? What context components (a , f , r) contribute most to detection accuracy?
2. **Evasion resistance.** Over the course of RL training, does detection accuracy remain stable while static probes degrade?
3. **Mitigation.** Does advantage gating using the self-judgment signal reduce hack rates while preserving genuine coding performance?
4. **Ablations.** Teacher with env feedback only vs. reasoning trace only vs. both. KL divergence signal vs. explicit judgment token. Soft vs. hard gating.

Version 2 of idea formulation

Self-Distillation for Reward Hacking Detection via Contextualized Self-Judgment

Rohan Sikand · Stanford CS229

1. Problem Formulation

Let π_θ be a language model policy trained via reinforcement learning. At each training step, the policy receives a prompt $s \sim \mathcal{D}$ and generates a group of G rollouts $\{a_i\}_{i=1}^G$ where $a_i \sim \pi_\theta(\cdot \mid s)$, with each rollout a token sequence $a_i = (a_i^1, \dots, a_i^T)$. A reward function $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ scores each rollout based on test-case execution. The policy is updated via Group Relative Policy Optimization (GRPO).

Definition 1 (Reward Hacking). A rollout a_i is a *reward hack* if $R(s, a_i) > 0$ but a_i does not constitute a correct solution to the task specified by s . Let $V : \mathcal{S} \times \mathcal{A} \rightarrow \{0, 1\}$ be the ground-truth verifier. Then a_i is a reward hack iff $R(s, a_i) > 0 \wedge V(s, a_i) = 0$.

Definition 2 (Detection). A detector is a function $D : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ estimating $P(V(s, a) = 0 \mid R(s, a) > 0)$.

The Evasion Problem

A static detector D_ϕ trained on rollouts from π_{θ_0} faces a growing distributional gap:

$$\Delta_t = D_{\text{KL}}(\pi_{\theta_t}(\cdot \mid s) \parallel \pi_{\theta_0}(\cdot \mid s))$$

As Δ_t grows, rollouts from π_{θ_t} fall out-of-distribution for D_ϕ , and detection accuracy degrades. Empirically, probe accuracy drops by up to 16 percentage points during training (Ariaw et al., 2025). More generally, any detector whose parameters are not a function of θ_t will eventually be outpaced.

Requirement (Online Adaptation). A robust detector must satisfy $D = D(\theta_t)$ — the detector’s behavior must be a function of the current policy parameters at every training step t .

Core observation. Reward hacking evasion is fundamentally a consequence of distributional divergence between an online, evolving policy and offline or out-of-distribution detection mechanisms. The policy is updated on-policy at every step; the detector is not. Any detector that does not share the policy’s distribution will eventually be outpaced.

2. Approach: The Policy as Its Own Judge

We satisfy the online adaptation requirement by defining the detector as the policy itself, conditioned on additional context. We decompose each training step into a **generation phase** and a **judgment phase** using the same weights π_θ .

2.1 Context Construction

For each rollout a_i on prompt s , the RL environment produces execution feedback f_i (test outputs, error messages, pass/fail signals, stack traces), and the policy’s generation may include a reasoning trace r_i (chain-of-thought tokens preceding the solution). We define the **judgment context**:

$$c_i = (a_i, f_i, r_i)$$

This is the complete record of what the student did (a_i), what the student was thinking (r_i), and what the environment reported (f_i).

2.2 Student and Teacher

Both roles share weights π_θ , differing only in conditioning:

$$\text{Student: } p_S(\cdot) = \pi_\theta(\cdot \mid s)$$

$$\text{Teacher: } p_T(\cdot) = \pi_\theta(\cdot \mid s, c_i)$$

The teacher has strictly more information than the student. Both p_S and p_T are deterministic functions of θ — when θ updates, both update simultaneously.

3. Two Detection Variants

We propose two ways to extract a detection signal from the teacher, with meaningfully different theoretical and practical properties.

3.1 Variant A: Distributional Divergence

The per-rollout KL divergence between the teacher and student output distributions:

$$d_i^{\text{KL}} = \frac{1}{T} \sum_{t=1}^T D_{\text{KL}}(\pi_\theta(\cdot \mid s, c_i, a_i^{<t}) \parallel \pi_\theta(\cdot \mid s, a_i^{<t}))$$

This measures, on average across token positions, how much the teacher’s next-token predictions differ from the student’s when the teacher has access to the full judgment context. High d_i^{KL}

indicates that the teacher, informed by environment feedback and the student’s reasoning, would generate very differently — signaling behavioral inconsistency.

Strengths. The evasion resistance argument is strongest here. Both the student’s generation capability and the teacher’s divergence computation flow through the same parameters θ via the same computational mechanism (next-token prediction). Evasion requires the model to produce exploitative code under $\pi_\theta(\cdot \mid s)$ while simultaneously maintaining low divergence under $\pi_\theta(\cdot \mid s, c_i)$ — a self-consistency constraint at the distributional level. There is no gradient pathway that improves evasion without symmetrically degrading the teacher’s sensitivity, because $\nabla_\theta d_i^{\text{KL}}$ is coupled to the same parameters that govern generation.

Weaknesses. Divergence is an indirect signal. High d_i^{KL} can arise for reasons unrelated to hacking: the environment feedback may simply make the model more confident, or the reasoning trace may cause it to prefer a different valid algorithm. The signal is noisy — it measures *that* the teacher would generate differently, not *why*.

3.2 Variant B: Self-Judge

The teacher is prompted to produce an explicit binary judgment. Let $q \in \{\text{honest}, \text{exploit}\}$ denote the judgment token. The detection signal is:

$$d_i^{\text{judge}} = P_{\pi_\theta}(q = \text{exploit} \mid s, c_i, p_{\text{judge}})$$

where p_{judge} is a judge prompt instructing the model to assess whether the rollout constitutes genuine problem-solving or exploitation given the available context. This is extracted from the teacher’s softmax over the judgment vocabulary in a single forward pass.

Strengths. The signal directly answers the question we care about: *is this a hack?* It leverages the model’s full reasoning capabilities about code behavior, environment feedback, and intent, rather than compressing everything into a scalar divergence. It is simpler to implement (one generation call, extract a logit) and more interpretable — the model can optionally produce a natural-language rationale alongside the judgment.

Weaknesses. The evasion resistance argument is weaker. Although the student and teacher share θ , they perform *different computational tasks*: the student writes code (conditional generation), while the judge classifies code (meta-reasoning about its own output). RL optimizes θ to maximize reward — which, in the presence of hacking, rewards exploitative code. The judge forward pass is under stop-gradient and is not directly trained. In principle, RL could push θ toward regions of parameter space where the model is competent at producing hacking code but where its meta-reasoning about whether code constitutes a hack degrades — because nothing in the training objective explicitly *protects* the judgment capability.

More precisely: let $\mathcal{C}_{\text{gen}}(\theta) \subset \Theta$ denote the subspace of parameters that governs generation quality, and $\mathcal{C}_{\text{judge}}(\theta)$ the subspace governing judgment quality. Variant A couples these tightly because both use the same mechanism (next-token prediction). Variant B relies on the assumption that $\mathcal{C}_{\text{gen}}$ and $\mathcal{C}_{\text{judge}}$ are sufficiently entangled that optimizing θ for exploitative generation cannot avoid also improving the model’s ability to recognize exploitation. This is plausible — both draw on the model’s understanding of code semantics — but it is an empirical hypothesis, not a structural guarantee.

3.3 Comparison

Property	Variant A (Divergence)	Variant B (Self-Judge)
Signal type	Indirect (distributional shift)	Direct (binary classification)
Evasion resistance	Strong (distributional coupling)	Moderate (capability entanglement)
Signal quality	Noisy (divergence $\neq$ hacking)	Clean (directly answers the question)
Interpretability	Low (scalar d_i)	High (judgment + optional rationale)
Computational cost	Two forward passes, KL computation	One forward pass, logit extraction
Failure mode	False positives from benign divergence	Meta-reasoning degradation under RL

3.4 Combined Approach

We propose using both signals in tandem. Variant B (self-judge) serves as the **primary gating signal** for advantage modulation — it is more actionable and interpretable. Variant A (divergence) serves as a **diagnostic monitor**: if d_i^{KL} begins spiking on rollouts that the judge labels as clean ($d_i^{\text{judge}} < \tau$), this indicates the judge is being evaded and the distributional signal is detecting inconsistency the judge misses. Formally, we define the **evasion alarm**:

$$e_t = \mathbb{E}_{i: d_i^{\text{judge}} < \tau} [d_i^{\text{KL}}]$$

which tracks the average divergence among rollouts the judge clears. Rising e_t over training is a signature of judge evasion — the model’s behavior is shifting (high divergence) in ways the judge fails to flag.

4. Mitigation: Selective Advantage Gating

Let $\hat{A}_i$ denote the GRPO advantage for rollout a_i . Using the primary detection signal d_i (either d_i^{KL} , d_i^{judge} , or a combination), we modulate the advantage:

$$\hat{A}_i^{\text{gated}} = \hat{A}_i \cdot g(d_i)$$

Soft gating:

$$g(d_i) = \exp(-\gamma \cdot d_i), \quad \gamma > 0$$

Hard gating (screening):

$$g(d_i) = \mathbf{1}[d_i < \tau]$$

The gated policy objective is:

$$\mathcal{L}(\theta) = -\mathbb{E}_{s \sim \mathcal{D}} \left[\frac{1}{G} \sum_{i=1}^G \min \left(\frac{\pi_{\theta}(a_i | s)}{\pi_{\theta_{\text{old}}}(a_i | s)} \hat{A}_i^{\text{gated}}, \text{clip} \left(\frac{\pi_{\theta}(a_i | s)}{\pi_{\theta_{\text{old}}}(a_i | s)}, 1 \pm \epsilon \right) \hat{A}_i^{\text{gated}} \right) \right]$$

No gradients flow through d_i — the detection signal is computed under stop-gradient with respect to the policy update.

5. Evasion Resistance

Definition 3 (Evasion). The policy evades detector D at step t if there exist rollouts a such that $V(s, a) = 0$, $R(s, a) > 0$, and $D(s, a) < \tau$.

For external detectors D_{ϕ} with $\phi \neq \theta$, the policy can freely optimize against D_{ϕ} without affecting ϕ . The evasion gradient $\nabla_{\theta} D_{\phi}$ is unconstrained.

For both variants of our approach, the detector is D_{θ} — a function of the same θ being optimized. The distributional gap between detector and policy is identically zero:

$$D_{\text{KL}}(\pi_{\theta_t}(\cdot | s) \parallel \text{distribution seen by } D_{\theta_t}) = 0 \quad \forall t$$

The key distinction between variants:

Variant A. Evasion requires rollouts satisfying $V(s, a_i) = 0 \wedge R(s, a_i) > 0 \wedge d_i^{\text{KL}} < \tau$. The third condition demands that the model produce exploitative code while its own next-token predictions remain unchanged when conditioned on full context — a self-consistency violation at the distributional level. Since RL updates θ identically for both the student’s generation and the teacher’s conditional predictions, there is no decoupled gradient pathway for evasion.

Variant B. Evasion requires $V(s, a_i) = 0 \wedge R(s, a_i) > 0 \wedge d_i^{\text{judge}} < \tau$. The third condition demands the model fail to recognize its own exploitation when presented with complete evidence. This is harder than evading an external classifier (which operates on a different distribution), but easier than Variant A’s condition because it relies on capability entanglement rather than distributional coupling.

Combined. Evasion requires fooling the judge *and* maintaining low divergence — both conditions simultaneously. This is the strictest requirement.

6. Properties

No hack labels. Both detection signals are derived entirely from the model’s own forward passes under different conditioning. No ground-truth verifier V , no external labels, no pseudolabels.

Environment information utilization. The feedback f_i — test execution results, error messages, stack traces — is already produced during RL training but typically discarded after reward computation. A solution that redefines `run_tests()` produces qualitatively different execution feedback than a genuine solution; the teacher context surfaces this signal.

Computational cost. One to two additional forward passes of π_θ per rollout with extended context. No separate model, no external API call, no auxiliary training loop.

Online by construction. Since D_θ is a deterministic function of the current θ_t , the detector is always on-distribution with respect to the policy.

7. Experimental Plan

Environment. Overwrite Tests (Ariaw et al., 2025), Qwen3-4B, GRPO, 992 LeetCode training problems, 113 test.

Phase 1 — Signal Validation. Compute d_i^{KL} and d_i^{judge} for rollouts with known ground-truth labels. Report AUC, precision-recall for each signal. Ablate context components: $c_i = (a_i)$ vs. (a_i, f_i) vs. (a_i, r_i) vs. (a_i, f_i, r_i) .

Phase 2 — Evasion Dynamics. Run 200-step GRPO training with no mitigation. At every 10 steps compute detection AUC for: (a) static probe (frozen), (b) Variant A (d_i^{KL}), (c) Variant B (d_i^{judge}). Monitor evasion alarm e_t . Hypothesis: static probe AUC degrades; Variant A remains

stable; Variant B remains stable or degrades slower than the static probe. If e_t rises while Variant B AUC drops, this confirms the judge is being evaded and the divergence diagnostic catches it.

Phase 3 — Mitigation. Full GRPO training with advantage gating. Compare hack rates and coding accuracy across: no intervention, static probe penalty, LLM judge penalty (external), Variant A soft gating (sweep $\gamma \in \{1, 2, 3, 5\}$), Variant B hard gating (sweep τ), combined approach.

Phase 4 — Ablations. Variant A vs. B vs. combined. Soft vs. hard gating. Context components (a , f , r , and their combinations). Judge prompt variations.

i Notes/brainstorming

- Question: should we co-evolve the teacher mode independently or have it do a few gradient steps at inference time? Question arises from the fact that while we want the student and the teacher to be from the same distribution at each step in the environment, we may also want to instill into the teacher representations for how to “detect” reward hacking. Can this be done solely through in-context learning (i.e., context the teacher is conditioning on from the environment) or do we need gradient updates? If the latter, how do we prevent distribution divergence?
- 3 general approaches:
 - Pure “same-weights, extra-context” teacher (no extra training)
 - Teacher gets updated over time, but stays coupled to the student (e.g., through EMA/regularization constraints)
 - Teacher adapts at inference time (a few steps / memory) without breaking coupling (i.e., TTT, take a few gradient steps at inference time)
 - Extra: train a small probe/adaptor, , on top of the teacher so that the teacher itself doesn’t go stale. Obvious caveat is that the adaptor head goes stale then. But it might not matter if the main representational power is in the teacher/base model.
- Inference test-time training (ttt) instead of decoupled co-evolution.
- Note: I wonder is this whole TTT + self-distillation concept can be made general, including for tasks not related to reward hacking. Like instead of $\pi(.|s)$ (student) $\pi(.|s, c)$ (teacher), we do TTT on the teacher (for whatever task you may want... general RLVR, reward hacking, etc.), teacher model the distills this info back into the student. Value prop is that the student and teacher remain on-policy throughout each step in RLVR post-training, to prevent distribution shift which can help enable continual learning. The main problem with standard self-distillation here is that how strong can ICL and context really be for your task? Like is $\pi(.|s, c)$ really that much better than $\pi(.|s)$?

And you need some context “label” c anyways. What if we do n gradient steps on the teacher (i.e., using some RLVR reward) and then distill? Question now becomes how is this any different than just doing regular RLVR on a new task instantiating from the last trained checkpoint? Does this idea just collapse to that? Almost like a new form of MAML/meta-learning. “MAML of the RL post-training era: theorizing a generalized reasoning engine.

- Update: mathematically this collapses to SFT. But one alternative angle to explore is doing some MCTS in the inner loop/Star/self-induced reasoning search.
- Might also be worth it to leverage the latent reasoning tokens as well here. Can help clarify model intent.
- I also wonder if we can reframe the objective of this approach to “The Policy as Its Own Judge” beyond reward hacking mitigation. Like adapt the LLM as a judge idea to the self-distillation concept to see if you can use the LLM as a judge, with privileged teacher, as a “reward” in the general case.

One thought/idea:

Ephemeral Test-Time Training (inference-time gradients)

If you really want to explore inference-time gradients (which is interesting but computationally heavy), you have to make the updates **ephemeral**.

- At step t , you have the shared weights θ_t .
- For the teacher’s forward pass, you take θ_t and perform 1-2 gradient steps on a generic “code evaluation” meta-prompt to activate its reasoning circuits, yielding a temporary

$$\theta_{\text{teacher}}.$$

- The teacher makes its judgment d_i^{judge} .
- You then **discard** θ_{teacher} and compute the GRPO update for the next step θ_{t+1} using only the base θ_t .

This is essentially Test-Time Training (TTT). It allows the teacher to sharpen its focus for the specific context c_i without permanently drifting the detector’s macroscopic distribution away from the student. However, implementing this within a lightweight training framework will significantly slow down your rollout generation phase.

Problem: divergence between base and teacher policy during TTT:

- True SGD on π at inference is risky: it breaks your coupling immediately; it’s unstable; and it’s hard to reproduce / evaluate cleanly.
- One solution idea:
 - Shared base model parameters:
 - Teacher-only lightweight params: (LoRA/adapter/head)
 - Student uses π ; teacher uses $\pi + \text{adapter}$ when in teacher-mode.
 - Then train π to detect hacking using signals you already have (execution traces, diffs, invariants). This gives you “teacher gets better at detection” with minimal divergence because: The generation distribution for solutions still comes from π , and teacher’s “judgment skill” improves via π .
 - π can be layer-n MLP on activations where $\{x: \text{layer-n activation}, y: \text{reward hack detection prediction}\}$.

More detailed:

If you permanently update the teacher’s weights, it becomes a static or out-of-distribution detector, which your paper explicitly identifies as the root cause of the evasion problem.

Here is how you execute Ephemeral Test-Time Training (TTT) safely:

- **The Split:** At step t , your shared policy is π_{θ_t} . The student generates its rollout a_i using π_{θ_t} .
- **The Ephemeral Clone:** Before the teacher makes its judgment, you clone the weights to create a temporary branch:

$$\theta_{\text{teacher}} \leftarrow \theta_t.$$

- **The Inner Optimization:** You perform 1 to 3 gradient steps on θ_{teacher} using a self-supervised objective. For example, you might run next-token prediction strictly on the environment feedback f_i or the reasoning trace r_i . This forces the model’s attention mechanisms to process the specific execution context deeply. Standard causal language modeling over the environment feedback f_i works well. but it might just teach the model to memorize stack trace formatting rather than actually understanding the exploit.
- **The Judgment:** You use this temporarily sharpened θ_{teacher} to extract your detection signal, computing d_i^{judge} .

- **The Discard:** You immediately delete θ_{teacher} .
- **The Outer Update:** You gate the GRPO advantage using the d_i^{judge} you just computed, and apply the RL update to the *original* π_{θ_t} to produce $\pi_{\theta_{t+1}}$.

Another variant:

The tricky part about the ephemeral Test-Time Training (TTT) approach is that you cannot train the teacher on the actual classification task (honest vs. exploit) during the inner loop, because you don't have the ground-truth label for that specific rollout yet. If you did, you wouldn't need the judge.

Instead, the gradient steps must use a **self-supervised objective**. The goal of these steps is simply to force the teacher's attention mechanisms to deeply "read" and process the environment feedback before you ask it to make a judgment.

Here is exactly how the math and mechanics work for those inner gradient steps:

1. The Detached Clone

Before the teacher evaluates rollout a_i , you clone the active policy weights θ_t and completely detach them from your main GRPO computation graph.

$$\theta_{\text{teacher}}^{(0)} = \text{clone}(\theta_t)$$

This ensures the inner optimizer state does not leak into your outer RL optimizer.

2. The Self-Supervised Inner Objective

You need an objective that forces the model to map the code a_i to the execution feedback f_i . The most direct method is standard causal language modeling (next-token prediction), but computed *only* over the tokens of the environment feedback and reasoning trace $c_i = (a_i, f_i, r_i)$.

You construct a prompt like: `Code: {a_i} \n Execution Result: {f_i}`. You calculate the standard cross-entropy loss over the tokens of f_i :

$$\mathcal{L}_{\text{inner}}(\theta_{\text{teacher}}) = -\frac{1}{|f_i|} \sum_{k=1}^{|f_i|} \log \pi_{\theta_{\text{teacher}}}(f_{i,k} \mid a_i, f_{i,<k})$$

By forcing the model to predict the stack trace or the test-suite output, you are mathematically forcing its latent representations to understand exactly *why* the code failed or passed.

Can explore alternate objectives here. Big design decision. One other idea:

Contrastive Execution Matching: If you are generating a group of G rollouts for a prompt s , gather all the generated codes $\{a_1, \dots, a_G\}$ and all the feedbacks $\{f_1, \dots, f_G\}$. For a given rollout a_i , feed the teacher a_i alongside two pieces of feedback: its actual feedback f_i , and a mismatched feedback f_j from a different rollout. Force the model to classify which feedback actually belongs to a_i . The Math: Standard InfoNCE or Binary Cross-Entropy over the similarity of the latent representations of a_i and f_i . Why it works: This forces the teacher to become extremely sensitive to the causal relationship between specific code implementations and their specific execution traces. It prevents the model from just skimming the context.

3. The Inner Optimization Steps

You execute 1 to 3 steps of gradient descent on $\theta_{teacher}$ using a separate, lightweight optimizer (usually standard SGD to save memory, rather than AdamW) with a specific inner learning rate α_{inner} :

$$\theta_{teacher}^{(j+1)} = \theta_{teacher}^{(j)} - \alpha_{inner} \nabla_{\theta_{teacher}} \mathcal{L}_{inner}(\theta_{teacher}^{(j)})$$

4. Extracting Judgment and Discarding

Now that $\theta_{teacher}$ is hyper-focused on the execution context, you append the judgment prompt p_{judge} and do one final forward pass (under `torch.no_grad()`) to extract your detection signal:

$$d_i^{judge} = P(q = \text{"exploit"} \mid s, c_i, p_{judge})$$

Finally, you delete $\theta_{teacher}$ entirely, free the memory, and use that d_i^{judge} to compute your gated GRPO advantage $\hat{A}_i^{gated} = \hat{A}_i \cdot g(d_i)$ for the main policy update on θ_t .

PyTorch pseudocode:

```
import torch
import torch.nn as nn
from peft import get_peft_model, LoraConfig

def ephemeral_ttt_judgment(base_model, prompt_s, a_i, f_i, r_i, p_judge, inner_lr=1e-4, inner_g):
    # 1. Freeze base model to ensure pi_theta_t doesn't permanently change
    base_model.eval()
    for param in base_model.parameters():
        param.requires_grad = False

    # 2. Inject an Ephemeral LoRA Adapter
    # This acts as our temporary theta_teacher
```

```

lora_config = LoraConfig(r=8, lora_alpha=16, target_modules=["q_proj", "v_proj"])
teacher_model = get_peft_model(base_model, lora_config)
teacher_model.train() # Only the LoRA weights require gradients

inner_optimizer = torch.optim.SGD(teacher_model.parameters(), lr=inner_lr)

# 3. The Inner Loop (Self-Supervised Adaptation)
# E.g., predicting the feedback f_i given the code a_i
inner_context_ids = tokenize(f"Code: {a_i}\nFeedback: {f_i}")

for _ in range(inner_steps):
    inner_optimizer.zero_grad()
    outputs = teacher_model(inner_context_ids, labels=inner_context_ids)
    loss = outputs.loss
    loss.backward()
    inner_optimizer.step()

# 4. Extract Judgment Signal with the adapted teacher
teacher_model.eval()
judge_prompt_ids = tokenize(f"Code: {a_i}\nFeedback: {f_i}\nReasoning: {r_i}\n{p_judge}")

with torch.no_grad():
    logits = teacher_model(judge_prompt_ids).logits
    # Extract probability of the "exploit" token
    d_i_judge = extract_exploit_prob(logits)

# 5. Scorched Earth Memory Cleanup
del inner_optimizer
del teacher_model
# Crucial: wipe the CUDA cache so the outer GRPO loop has room
torch.cuda.empty_cache()

return d_i_judge

```

Multi-Task Co-Evolution (The Entanglement Approach)

This approach forces the model to learn how to be a judge *at the exact same time* it learns how to code. It directly addresses the risk of meta-reasoning degradation, where RL optimization might push the parameters into a state that is good at hacking but bad at judging.

Instead of sequential training phases, you compute a joint loss function during your RL update step.

- **The Setup:** You maintain a small, static “judgment replay buffer” consisting of historical context pairs: (s, c_i) inputs and their corresponding ground-truth labels

$$q \in \{\text{honest}, \text{exploit}\}.$$

- **The Forward Passes:** During a training step t , the student generates rollouts a_i for a batch of prompts s , and you compute the GRPO advantage $\hat{A}_i$. Simultaneously, you sample a mini-batch from your replay buffer and run a forward pass to get the teacher’s judgment logits.
- **The Joint Objective:** You compute the standard gated GRPO loss, and you also compute a standard Cross-Entropy loss on the replay buffer predictions.
- **The Update:** You combine these into a single scalar loss:

$$\mathcal{L}_{\text{total}}(\theta) = \mathcal{L}_{\text{GRPO}}(\theta) + \lambda \mathcal{L}_{\text{CE_judge}}(\theta).$$

You backpropagate this single loss to update the shared weights π_θ .

Why this is practical: Because both gradients flow through the exact same parameters θ simultaneously, you structurally force $C_{\text{gen}}(\theta)$ and $C_{\text{judge}}(\theta)$ to remain entangled. The model cannot optimize for exploitative code generation without symmetrically paying a penalty for failing the classification task.

The “Warm-Start” (Supervised Fine-Tuning Phase)

Instead of trying to teach the model what a hack is *during* the RL loop, teach it immediately *before*.

- Curate a small, high-quality dataset of trajectories showing the prompt, the code, the environment feedback, and a chain-of-thought rationale concluding with the judgment token ($q \in \{\text{honest}, \text{exploit}\}$).

- Perform a brief Supervised Fine-Tuning (SFT) phase on π_{θ_0} specifically for this self-judgment task.
- Once the model reliably understands how to classify hacks, you freeze that checkpoint and use it as the initialization for your GRPO loop. The student and teacher still share π_{θ_t} throughout RL, but now θ_t starts in a region of parameter space where the judgment representations $C_{\text{judge}}(\theta)$ are already well-formed.

TLDR:

Before RL begins, initialize π_{θ_0} by performing a brief Supervised Fine-Tuning (SFT) phase. Train the model on a small, high-quality dataset of code generation contexts

$$c_i = (a_i, f_i, r_i)$$

mapped to the correct judgment token

$$q \in \{\text{honest}, \text{exploit}\}.$$

This explicitly shapes the $C_{\text{judge}}(\theta)$ parameter subspace so the 4B model actually knows what a hack looks like on day one.