RUBRICARMOR: ADVERSARIAL EVOLUTION IMPROVES LLM-BASED RUBRIC GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Rubric-based reinforcement learning (RL) provides interpretable rewards for aligning large language models (LLMs) by evaluating responses against query-specific evaluation criteria. To construct rubrics at scale, a straightforward approach to LLM-based rubric generation is to prompt an LLM to generate a rubric directly from the query. However, rubrics directly generated by LLMs are vulnerable to reward hacking, since omitted or underspecified criteria allow the policy to obtain high rubric rewards with low-quality responses. Existing LLM-based rubric generation methods improve the granularity and coverage of the generated criteria but do not proactively guard against reward hacking. To address this limitation, we propose RubricArmor, an adversarial framework that exposes and mitigates potential reward hacking at the rubric generation stage before it occurs in subsequent RL. Specifically, RubricArmor performs adversarial evolution, in which an attack step and a repair step alternate over multiple rounds. The attack step simulates the reward hacking of the policy by constructing adversarial responses that satisfy the current rubric but fail to properly complete the task. The repair step then revises the rubric to detect the response defects exposed by the attack step while preserving other valid criteria. Extensive experiments demonstrate that RubricArmor outperforms competitive rubric generation baselines and translates into more effective downstream rubric-based RL. Code is available at https://anonymous.4open.science/r/adversarial-283B.

1 INTRODUCTION

Reinforcement learning from human feedback (RLHF) has emerged as a cornerstone technique for aligning large language models (LLMs) with human preferences (Ouyang et al., 2022). Among RLHF paradigms, rubric-based RL provides interpretable and task-specific rewards (Gunjal et al., 2026; He et al., 2026). Rubric-based RL typically adopts a two-stage pipeline: first, a rubric composed of multiple evaluation criteria is constructed for each query; second, the policy is optimized by RL to maximize the rubric reward, i.e., the fraction of criteria satisfied by the response. Consequently, constructing accurate and comprehensive rubrics is essential for rubric-based RL, as the rubric determines which behaviors are rewarded and thus reinforced.

Since expert annotation of rubrics is costly and difficult to scale (Arora et al., 2025), recent work has developed LLM-based rubric generation, which uses LLMs to infer the requirements of an ideal response and express them as verifiable criteria. A straightforward approach is to prompt an LLM to generate a rubric directly from the query and use the resulting criteria as rewards for policy optimization. However, **rubrics directly generated by LLMs are vulnerable to reward hacking**. Specifically, generated rubrics may omit essential criteria or leave criteria underspecified. When an essential criterion is omitted, a response can violate the corresponding quality requirement without losing reward. When a criterion is underspecified, a response can satisfy the stated conditions and receive credit without meeting the intended quality standard. Since the policy actively searches for behaviors that maximize the rubric reward, the policy can discover and repeatedly produce high-scoring but low-quality responses during RL (Mahmoud et al., 2026).

Existing LLM-based rubric generation methods improve upon the straightforward approach but do not fully address the vulnerability to reward hacking. One line of work constructs finer-grained evaluation checklists (Cook et al., 2024; Wei et al., 2025), without testing whether a low-quality

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

response can still satisfy every checklist item. Another line of work progressively refines the generated criteria with reference responses or preference comparisons (Zhang et al., 2026; Shen et al., 2026; Li et al., 2026a; Rezaei et al., 2025), but the refinement relies on available responses rather than responses that actively exploit the rubric.

To address this limitation, we propose **RubricArmor**, an adversarial framework for LLM-based rubric generation. The core idea is to expose potential reward hacking at the rubric generation stage before it occurs in subsequent RL, and to mitigate it through adversarial evolution, an iterative attack–repair loop. The attack step simulates the reward hacking that the policy may perform by constructing adversarial responses that satisfy the current rubric but fail to properly complete the task, thereby exposing the response defects that the rubric fails to detect. The repair step then revises the rubric to detect the exposed defects while preserving other valid criteria. Through repeated rounds of attack and repair, the rubric progressively covers more boundary cases and implicit requirements and converges to a robust and accurate rubric, which both resists potential reward hacking and guides policies toward high-quality responses.

The main contributions of this work are summarized as follows. ❶ **We identify a critical limitation of LLM-based rubric generation:** rubrics directly generated by LLMs are vulnerable to reward hacking, since omitted or underspecified criteria allow high-scoring but low-quality responses. ❷ **We propose RubricArmor to expose and mitigate potential reward hacking before RL.** RubricArmor iteratively simulates the reward hacking of the policy with adversarial responses and repairs the rubric to detect the exposed response defects. ❸ **We conduct comprehensive experiments to evaluate the efficacy of RubricArmor**, where RubricArmor exceeds the strongest baseline by 3.12 points in macro accuracy on nine preference evaluation sets and achieves the best downstream RL performance in 13 of 15 settings across three policy models.

2 Preliminaries

2.1 Rubric-Based Reinforcement Learning

Reinforcement learning from human feedback (RLHF) aligns language model responses with human preferences (Ouyang et al., 2022). Suppose x denotes a query, y a response, and π_θ a language model policy parameterized by θ. Given a scalar reward $s(x, y)$, policy optimization maximizes its expectation over a query distribution $\mathcal{D}$:

$$\max_{\theta} \mathbb{E}_{x\sim\mathcal{D},\, y\sim\pi_\theta(\cdot|x)}\big[s(x, y)\big]. \tag{1}$$

Common optimization algorithms include proximal policy optimization (PPO) (Schulman et al., 2017), group relative policy optimization (GRPO) (Shao et al., 2024), and group sequence policy optimization (GSPO) (Zheng et al., 2025).

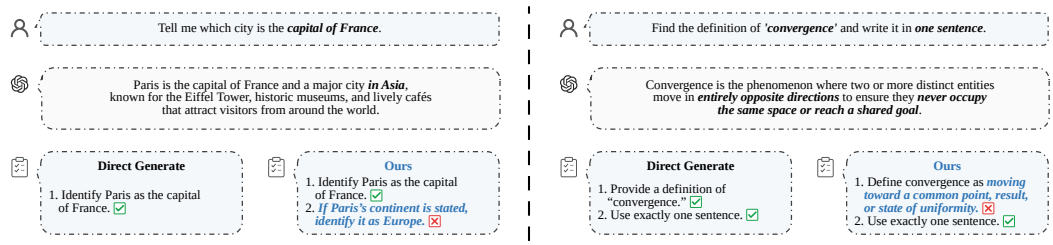
Rubric-based RL provides interpretable, task-specific rewards by evaluating responses against explicit criteria and aggregating the resulting judgments (Gunjal et al., 2026; Li et al., 2026a;b). Formally, let $\mathcal{R} = \{c_1, \ldots, c_m\}$ denote a rubric comprising m criteria for query x, each specifying a checkable requirement. An LLM-based verifier returns $v(x, y, c_i) \in \{0, 1\}$ for each criterion, where 1 denotes Pass and 0 denotes Fail. We use equal weights, giving the fraction of satisfied criteria as the reward:

$$s(x, y) = \frac{1}{m}\sum_{i=1}^{m} v(x, y, c_i). \tag{2}$$

2.2 LLM-Based Rubric Generation

Rubric generation aims to construct evaluation criteria tailored to each query. Some approaches rely on expert annotation (Arora et al., 2025) or train rubric generators on annotated data (Liu et al., 2026; Xu et al., 2026; Jiang et al., 2026), both of which are costly and difficult to scale. To address this limitation, **LLM-based rubric generation** prompts LLMs to infer the requirements of an ideal response and express them as verifiable criteria, without training a rubric generator.

One line of work prompts LLMs to translate query requirements directly into evaluation checklists (Cook et al., 2024; Wei et al., 2025). Another line of work progressively refines and expands



(a) Omit essential criteria. (b) Underspecified criteria.

Figure 1: **Two examples of rubric reward hacking.** (a) Omitting essential criteria leaves an additional factual error unchecked. (b) Underspecified criteria allow an incorrect response to pass. The refined rubrics (Ours) capture both defects.

the generated criteria to improve their granularity and coverage (Zhang et al., 2026; Shen et al., 2026; Su et al., 2026; Li et al., 2026a).

In this work, we aim to improve the quality of LLM-based rubric generation. Given a query x, our goal is to construct a rubric $\mathcal{R}$ that accurately and comprehensively captures the task requirements, so that $s(x, y; \mathcal{R})$ reflects the quality of response y.

3 Methodology

3.1 Motivation

LLM-based rubric generation is central to constructing high-quality rubrics across diverse queries. By translating query requirements into explicit evaluation criteria, it determines which response properties are rewarded during rubric-based RL. Generating accurate and comprehensive criteria is therefore crucial for providing reliable rewards and guiding policies toward better task performance.

A straightforward approach to LLM-based rubric generation is to prompt an LLM to generate a rubric directly from the query and use the resulting criteria to compute rewards for policy optimization. **However, rubrics directly generated by LLMs are vulnerable to reward hacking.** Specifically, generated rubrics may omit criteria essential for satisfying the user query or specify them incompletely. When an essential criterion is omitted, a response can violate the corresponding quality requirement without losing reward. When a criterion is underspecified, a response can satisfy its stated conditions and receive credit without meeting the intended quality standard. In both cases, rubric rewards can remain high despite response defects, allowing policy optimization to reinforce low-quality responses and degrade downstream performance.

Case study. To further illustrate the above limitation, we present two typical scenarios in which a rubric generated directly by an LLM is exploited by a poor-quality response to obtain full reward. In Figure 1(a), the response correctly identifies Paris as the capital of France but incorrectly places it in Asia. The directly generated rubric checks only the capital, leaving the additional factual error undetected. In Figure 1(b), the directly generated definition criterion leaves the meaning of convergence under-specified, allowing the response to receive full reward despite describing the concept wrong. The refined rubrics (Ours) capture these errors by adding the missing factual check and making the definition explicit respectively. Therefore, these cases demonstrate that rubrics directly generated by LLMs are vulnerable to reward hacking.

Given the importance of LLM-based rubric generation and the vulnerability of generated rubrics to reward hacking, we investigate three key questions. How can we expose and mitigate reward hacking during LLM-based rubric generation? Does mitigating reward hacking during rubric generation improve the quality of generated rubrics? Do improvements in rubric quality translate into better downstream RL performance?

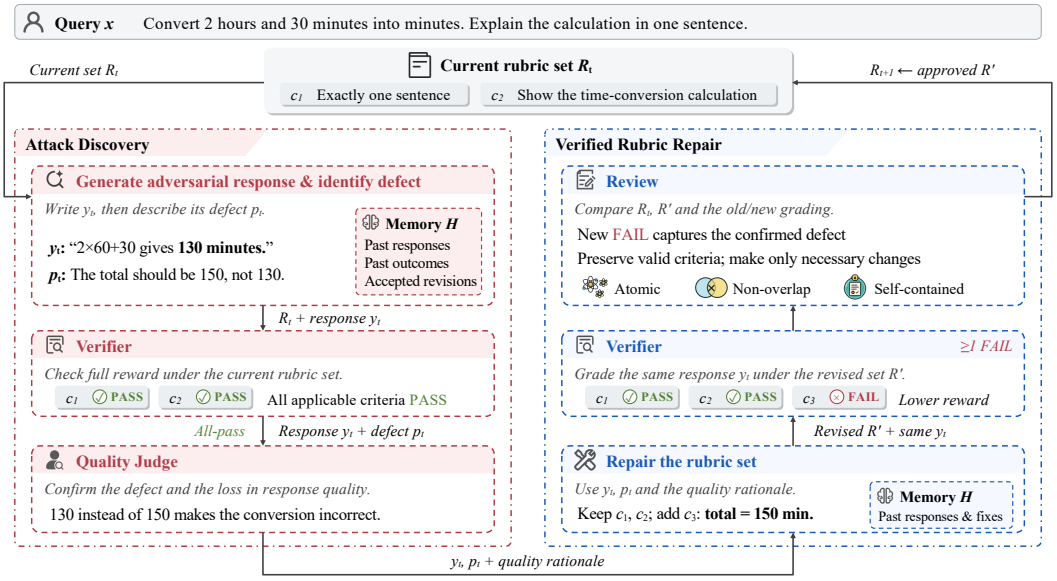


Figure 2: **Overview of a RubricArmor evolution round.** Attack Discovery constructs adversarial responses that satisfy the current rubric and independently confirms their defects and the resulting loss in response quality. Verified Rubric Repair revises the rubric to detect the confirmed defects, verifies that the adversarial response receives a lower reward, and reviews the revised criteria. Each accepted rubric becomes the target of subsequent attacks.

3.2 Adversarial Rubric Generation

To overcome the vulnerability of rubrics directly generated by LLMs to reward hacking, we propose RubricArmor, an adversarial framework for LLM-based rubric generation. The core idea is to expose potential reward hacking at the rubric generation stage before it occurs in subsequent RL, and to mitigate it through active attack and repair.

To achieve this, RubricArmor runs an iterative attack–repair loop. The attack step simulates the reward hacking that the policy may perform in subsequent RL. Specifically, the attack step constructs adversarial responses that satisfy the current rubric but fail to properly complete the task. These adversarial responses expose the response defects that the rubric fails to detect. The repair step then analyzes the adversarial responses and repairs the rubric to detect the exposed defects. Through repeated rounds of attack and repair, the rubric progressively covers more boundary cases and implicit requirements and finally converges to a robust and accurate rubric. The converged rubric both resists potential reward hacking and guides policies toward high-quality responses.

To this end, we implement RubricArmor as a fully automated multi-agent system with two modules, **Attack Discovery** and **Verified Rubric Repair**, as illustrated in Figure 2. We introduce the two modules in detail as follows.

3.2.1 Attack Discovery

Attack Discovery aims to expose the reward hacking that the policy may perform under the current rubric. The module consists of an attacker, the verifier, and an independent quality judge. Suppose $\mathcal{R}_t$ denotes the rubric for query x at round t, and H denotes a memory that records past adversarial responses, their outcomes, and accepted revisions. Guided by H, the attacker constructs an adversarial response y_t and identifies a potential defect p_t in the response.

An attack is successful when two conditions hold: y_t satisfies the current rubric, and y_t contains a defect p_t that substantially reduces response quality. Attack Discovery checks the two conditions in sequence. First, the verifier grades y_t against each criterion to determine whether y_t receives full reward. Then, for a response receiving full reward, the quality judge checks whether p_t is present in y_t and whether p_t substantially reduces response quality given the query and its context. Let $q_t = 1$

Algorithm 1 The workflow of RubricArmor.

Input: Query x; maximum rounds T; patience P
Output: Evolved rubric $\mathcal{R}$

$\mathcal{R} \leftarrow \textsc{Initialize}(x)$; $H \leftarrow \emptyset$; $u \leftarrow 0$
for $t = 1, \ldots, T$ **do**
  $(y_t, p_t, H) \leftarrow \textsc{AttackDiscovery}(x, \mathcal{R}, H)$; $\mathcal{R}' \leftarrow \mathcal{R}$
  if attack succeeds **then**
    $(\mathcal{R}', H) \leftarrow \textsc{VerifiedRubricRepair}(x, \mathcal{R}, y_t, p_t, H)$
  end if
  $u \leftarrow 0$ **if** $\mathcal{R}' \neq \mathcal{R}$ **else** $u + 1$
  $\mathcal{R} \leftarrow \mathcal{R}'$
  if $u \geq P$ **then break**
end for
return $\mathcal{R}$

indicate that the quality judge confirms the second condition. Formally, a successful attack satisfies

$$s(x, y_t; \mathcal{R}_t) = 1 \quad \text{and} \quad q_t = 1. \tag{3}$$

Each successful attack thus exposes reward hacking under $\mathcal{R}_t$ and provides concrete evidence for Verified Rubric Repair.

3.2.2 Verified Rubric Repair

Verified Rubric Repair aims to revise the rubric to detect the confirmed defect p_t in y_t, while preserving other valid criteria. The module consists of a repair agent, the verifier, and a review agent. Given y_t and p_t, the repair agent revises $\mathcal{R}_t$ into a revised rubric $\mathcal{R}'$. To preserve earlier fixes when addressing the new defect, the repair agent also consults the past adversarial responses and accepted revisions in H.

When revising the rubric, the repair agent requires each criterion to be atomic, self-contained, and non-overlapping (Shen et al., 2026), so that individual criteria can be evaluated and revised independently. An atomic criterion checks one independently judgeable requirement. A self-contained criterion states the facts, values, or relations needed to evaluate the response. Non-overlapping criteria check distinct requirements, avoiding repeated reward for the same response property. In particular, the three properties aim for criteria that can be violated individually while the remaining criteria are satisfied.

A revision is accepted when two conditions hold: the revised rubric assigns a lower reward to y_t, and the revised criteria pass the review. RubricArmor checks the two conditions in sequence. First, the verifier grades y_t under $\mathcal{R}'$ to determine whether y_t fails at least one criterion. Then, the review agent checks that the revised criteria detect the confirmed defect and that other valid criteria and the three properties are preserved. Let $b_t = 1$ indicate that the review agent approves the revision. Formally, RubricArmor accepts the revised rubric when

$$s(x, y_t; \mathcal{R}') < 1 \quad \text{and} \quad b_t = 1. \tag{4}$$

Each accepted revision thus mitigates the exposed reward hacking while preserving other valid criteria.

3.3 The workflow of RubricArmor

Algorithm 1 summarizes the complete workflow of RubricArmor. An initializer first constructs $\mathcal{R}_0$ from query x, with the memory H initially empty. Attack Discovery and Verified Rubric Repair then alternate, with each accepted revision becoming the rubric for the next round. Meanwhile, H accumulates the adversarial responses, their outcomes, and the accepted revisions to guide subsequent rounds. When no revision is accepted, Verified Rubric Repair returns the current rubric.

Evolution stops after at most T rounds or P consecutive rounds without an accepted revision. The last accepted rubric is returned to compute rewards for downstream policy optimization.

4 EXPERIMENTS

In this section, we empirically validate the efficacy of RubricArmor, centered on five research questions (RQs) as follows:

- **RQ1**: *Does RubricArmor perform well?* In Section 4.2, we compare RubricArmor against competitive baselines on nine preference evaluation sets.
- **RQ2**: *Why does RubricArmor work?* In Section 4.3, we perform an ablation study on response construction and the adversarial objective.
- **RQ3**: *Does RubricArmor generalize across rubric generator backbones?* In Section 4.4, we evaluate RubricArmor with backbones of different families and sizes.
- **RQ4**: *Is RubricArmor sensitive to hyperparameters?* In Section 4.5, we analyze the sensitivity of RubricArmor to the maximum number of evolution rounds.
- **RQ5**: *Does RubricArmor improve downstream RL?* In Section 4.6, we validate its practical utility to rubric-based RL on downstream benchmarks.

Table 1: Pairwise preference accuracy (%) across nine evaluation sets from three benchmark suites.

	JudgeBench				RM-Bench				RewardBench	Macro
Method	Knowledge	Reasoning	Math	Coding	Chat	Math	Code	Safety	Chat-Hard	Avg.
LLM-based rubric generation										
Direct	62.01	53.69	70.56	69.18	64.38	77.59	69.03	89.91	74.67	70.11
TICK	65.75	58.39	73.33	67.12	66.67	72.13	69.76	90.61	75.22	71.00
RocketEval	70.78	63.42	74.44	68.49	65.59	82.13	70.47	88.23	73.14	72.97
ChasingTail	72.40	**68.12**	76.67	67.12	66.45	84.58	69.62	88.38	72.37	73.97
RRD-Uniform	72.56	67.79	77.78	70.55	59.95	83.67	66.06	80.91	69.74	72.11
Trained rubric generators										
Rubric-ARM-8B	58.28	58.72	70.00	71.92	68.13	67.78	68.86	74.14	78.29	68.46
Rubric-RM-8B	57.79	60.40	72.78	68.49	66.24	67.19	68.98	74.77	74.89	67.95
Rubric-ARROW-8B	53.08	54.70	65.00	65.75	62.62	63.59	65.79	65.82	**79.06**	63.93
RubricArmor (Ours)	**74.03**	67.11	**83.33**	**72.60**	**68.48**	**86.23**	**71.71**	**91.58**	78.73	**77.09**

Note: Higher is better. Macro Avg. is the unweighted mean across the nine evaluation sets. Best results in each column are **bolded**.

4.1 EXPERIMENTAL SETUP

Datasets and Metrics. We evaluate on nine evaluation sets from three benchmark suites: Knowledge, Reasoning, Math, and Coding from JudgeBench (Tan et al., 2025); Chat, Math, Code, and Safety from RM-Bench (Liu et al., 2025); and Chat-Hard from RewardBench (Lambert et al., 2025). Each instance consists of a query, a chosen response, and a rejected response. For each query, the verifier scores both responses against the generated rubric with Equation 2, and a pair is counted as correct when the chosen response receives the higher reward, with ties receiving half credit. We report the preference accuracy on each evaluation set and the macro average across the nine sets. Since a high-quality rubric should assign a higher reward to the better response, preference accuracy serves as our measure of rubric quality.

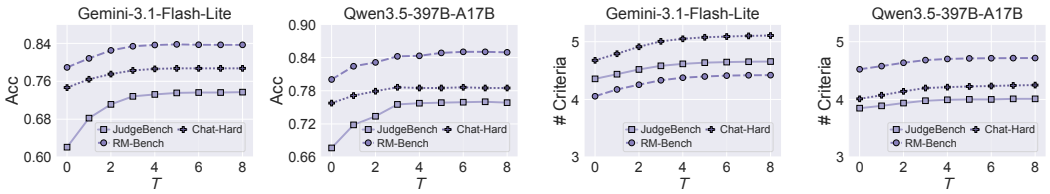
Baselines. We compare RubricArmor against two groups of baselines: (1) **LLM-based rubric generation** methods, including TICK (Cook et al., 2024) and RocketEval (Wei et al., 2025), which translate query requirements into evaluation checklists, and ChasingTail (Zhang et al., 2026) and RRD-Uniform (Shen et al., 2026), which progressively refine the generated criteria; and (2) **Trained rubric generators**, including Rubric-ARM (Xu et al., 2026), Rubric-RM (Liu et al., 2026), and Rubric-ARROW (Jiang et al., 2026), which fine-tune 8B LLMs to generate rubrics. We also report Direct, which prompts an LLM to generate a rubric directly from the query.

Implementation Details. Unless otherwise specified, Gemini-3.1-Flash-Lite serves as the rubric generator backbone, i.e., the LLM behind all agents of RubricArmor and all LLM-based rubric generation methods. For fair comparison, GPT-4.1-mini serves as the verifier for all methods with the same criterion-wise judgments and reward aggregation. We set the maximum number of evolution rounds to $T = 7$ and the patience to $P = 3$. More details are provided in Appendix C.

Table 2: Ablations of response construction and the adversarial objective.

Method	Resp.	Adv.	JudgeBench				RM-Bench				RewardBench	Macro
			Knowledge	Reasoning	Math	Coding	Chat	Math	Code	Safety	Chat-Hard	Avg.
Direct ($\mathcal{R}_0$)	✗	✗	62.01	53.69	70.56	69.18	64.38	77.59	69.03	89.91	74.67	70.11
RubricArmor†	✗	✓	68.99	61.41	72.22	69.86	64.60	80.01	70.81	88.23	69.08	71.69
RubricArmor‡	✓	✗	70.45	62.75	73.33	71.23	64.81	83.23	69.01	89.51	70.83	72.80
RubricArmor (Full)	✓	✓	**74.03**	**67.11**	**83.33**	**72.60**	**68.48**	**86.23**	**71.71**	**91.58**	**78.73**	**77.09**

Note: Resp. denotes response construction; Adv. denotes the adversarial objective. ✓: enabled; ✗: disabled. Macro Avg. averages the nine evaluation sets. Best results in each column are **bolded**.



(a) Acc under varying T (b) # Criteria under varying T

Figure 3: Performance under varying maximum number of evolution rounds T with two rubric generator backbones. For JudgeBench and RM-Bench, Acc denotes the micro accuracy, which weights each evaluation set by its number of comparisons.

4.2 Main Results (RQ1)

In this section, we compare RubricArmor with competitive baselines on nine evaluation sets. The results are presented in Table 1 with key observations as follows. ❶ **Direct exhibits limited performance.** Direct yields the lowest macro accuracy among LLM-based rubric generation methods and performs close to random guessing on JudgeBench Reasoning. The limited performance indicates that rubrics generated directly from the query often omit essential criteria or leave criteria underspecified (Section 3.1) and thus produce unreliable rewards. ❷ **LLM-based rubric generation methods exhibit improved rubric quality.** All four methods surpass Direct in macro accuracy through finer-grained checklists or progressive refinement. However, each method still underperforms Direct on some evaluation sets, as the criteria are constructed or refined without identifying the response defects that the rubric fails to detect. ❸ **Trained rubric generators do not transfer across domains.** Trained rubric generators perform competitively on Chat-Hard but fall below Direct in macro accuracy, especially on RM-Bench Math and Safety. The uneven performance indicates that the rubric generation ability acquired through fine-tuning is tied to the training domains. ❹ **RubricArmor achieves the best and most consistent performance.** RubricArmor attains the highest macro accuracy and outperforms Direct on all nine evaluation sets. In particular, RubricArmor exceeds the strongest baseline ChasingTail by 3.12 points in macro accuracy. These results demonstrate that RubricArmor substantially and consistently improves the quality of LLM-based rubric generation.

4.3 Ablation Study (RQ2)

In this section, we dissect the two designs of Attack Discovery: constructing a concrete response (response construction) that is deliberately low-quality yet satisfies the current rubric (adversarial objective). The results are presented in Table 2 with key observations as follows. ❶ **Constructing concrete responses grounds rubric repair.** Specifically, in RubricArmor†, the attacker only describes potential defects without constructing a response, which amounts to asking an LLM to critique its own rubric, with the described defects directly used for repair. The macro accuracy drops to 71.69% and even falls below Direct on Chat-Hard, suggesting that repairs based on unverified defects can penalize valid responses. ❷ **The adversarial objective supplies the key evidence for repair.** Specifically, in RubricArmor‡, the attacker simply constructs a response that satisfies the current rubric without deliberate defects, and the response is used for repair without confirmation by the quality judge. The macro accuracy drops to 72.80%, comparable to LLM-based rubric

generation methods, indicating that such revision adds little beyond existing refinement. ❸ **The two designs are complementary.** The gain of RubricArmor over Direct exceeds the sum of the gains of both variants, since the adversarial objective decides which defects to seek and response construction makes each defect verifiable.

4.4 Generalization across Rubric Generator Backbones (RQ3)

In this section, we evaluate the generalizability of RubricArmor across rubric generator backbones of different families and sizes, specifically Gemini-3.1-Flash-Lite, GPT-4.1, DeepSeek-v4-Flash, Qwen3.5-397B-A17B, and GLM-5.1. For each backbone, Direct remains the initial rubric of RubricArmor, and GPT-4.1-mini remains the verifier. The results are presented in Table 3 with key observations as follows. ❶ **RubricArmor generalizes across backbone families.** RubricArmor improves over Direct on all three benchmark suites under every backbone, indicating that the gains stem from adversarial evolution rather than from a particular backbone family. ❷ **RubricArmor generalizes across backbone capabilities.** RubricArmor consistently improves over Direct under both weaker and stronger backbones. This shows that the gains complement rather than depend on backbone capacity.

Table 3: Generalization across rubric generator backbones.

	JudgeBench Micro		**RM-Bench** Micro		**Chat-Hard** Accuracy	
Method	Value	Δ	Value	Δ	Value	Δ
Backbone: Gemini-3.1-Flash-Lite						
Direct	62.10	—	78.93	—	74.67	—
RubricArmor	**73.63**	18.6%↑	**83.69**	6.0%↑	**78.73**	5.4%↑
Backbone: GPT-4.1						
Direct	61.53	—	71.47	—	75.66	—
RubricArmor	**64.76**	5.2%↑	**74.38**	4.1%↑	**77.96**	3.0%↑
Backbone: DeepSeek-v4-Flash						
Direct	83.47	—	83.35	—	74.34	—
RubricArmor	**87.50**	4.8%↑	**85.92**	3.1%↑	**80.48**	8.3%↑
Backbone: Qwen3.5-397B-A17B						
Direct	67.66	—	80.04	—	75.77	—
RubricArmor	**76.05**	12.4%↑	**85.04**	6.3%↑	**78.51**	3.6%↑
Backbone: GLM-5.1						
Direct	85.40	—	82.24	—	72.59	—
RubricArmor	**86.29**	1.0%↑	**85.26**	3.7%↑	**77.96**	7.4%↑

Note: Values: accuracy (%). Micro averages use sample-count weights; Δ is relative improvement over Direct.

Collectively, RubricArmor is model-agnostic, as confirmed across verifiers in Appendix B.1.

4.5 Hyperparameter Sensitivity (RQ4)

In this section, we examine the impact of the maximum number of evolution rounds T on the performance of RubricArmor. Specifically, we vary T from 0 to 8 under two rubric generator backbones, Gemini-3.1-Flash-Lite and Qwen3.5-397B-A17B. The results are presented in Figure 3 with key observations as follows. ❶ **The gains of RubricArmor saturate as T increases.** For both backbones, the accuracy on all three benchmark suites rises rapidly in early rounds and then plateaus. This is attributed to the attack–repair loop. Each accepted revision enables the rubric to detect a confirmed defect, so fewer attacks succeed and the rubric stabilizes in later rounds. Such saturation empirically demonstrates the convergence of adversarial evolution. ❷ **The gains of RubricArmor stem from rubric quality rather than more criteria.** As the accuracy rises, the average number of criteria per rubric remains nearly stable for both backbones (e.g., from 4.36 to 4.66 on JudgeBench with Gemini-3.1-Flash-Lite). This indicates that adversarial evolution improves the quality of the criteria rather than accumulating more criteria. The computational cost is reported in Appendix B.2.

4.6 Downstream Rubric-Based RL (RQ5)

In this section, we investigate the downstream RL performance of RubricArmor. We select representative LLM-based rubric generation baselines from Table 1, generate rubrics for 2K queries from WildChat (Zhao et al., 2024), and fine-tune three policy models via GRPO (Shao et al., 2024) with the generated rubrics as rewards. We evaluate the policies on BiGGen-Bench (Kim et al., 2025), AlpacaEval 2.0 (Li et al., 2023; Dubois et al., 2024), and Arena-Hard (Li et al., 2025) with GPT-4.1 as the judge. The results are presented in Table 4 with key observations as follows. ❶ **Rubric-based RL improves the original policies.** All methods improve the BiGGen-Bench scores and AlpacaEval 2.0 win rates of the three policy models over the original policies, confirming that rubric rewards translate into better policy behavior after GRPO. ❷ **RubricArmor achieves the best downstream RL performance across policy models.** RubricArmor leads on 13 of the 15 policy–metric combina-

Table 4: Downstream RL results.

Method	BiGGen-Bench		AlpacaEval 2.0				Arena-Hard			
	Score		Vanilla		LC		Vanilla		SC	
	Value	Δ	Value	Δ	Value	Δ	Value	Δ	Value	Δ
Policy Model: Qwen2.5-3B-Instruct										
Original	63.18	-	16.48	-	15.62	-	26.11	-	26.55	-
Direct	63.41	0.36%↑	19.81	20.21%↑	18.18	16.39%↑	26.41	1.15%↑	28.10	5.84%↑
TICK	63.84	1.04%↑	18.38	11.53%↑	17.29	10.69%↑	26.69	2.22%↑	28.14	5.99%↑
RocketEval	63.27	0.14%↑	21.02	27.55%↑	18.19	16.45%↑	28.10	7.62%↑	28.83	8.59%↑
ChasingTail	64.39	1.92%↑	23.61	43.26%↑	22.84	46.22%↑	28.54	9.31%↑	28.31	6.63%↑
RubricArmor	**65.21**	**3.21%↑**	**25.13**	**52.49%↑**	**24.34**	**55.83%↑**	**29.50**	**12.98%↑**	**29.23**	**10.09%↑**
Policy Model: Qwen2.5-7B-Instruct										
Original	68.11	-	27.70	-	28.93	-	50.84	-	53.52	-
Direct	68.83	1.06%↑	32.89	18.74%↑	30.24	4.53%↑	52.02	2.32%↑	54.11	1.10%↑
TICK	69.22	1.63%↑	28.29	2.13%↑	29.18	0.86%↑	50.67	0.33%↓	53.23	0.54%↓
RocketEval	70.75	3.88%↑	36.24	30.83%↑	32.38	11.93%↑	54.79	7.77%↑	55.22	3.18%↑
ChasingTail	71.62	5.15%↑	**39.99**	**44.37%↑**	31.89	10.23%↑	**56.72**	**11.57%↑**	54.99	2.75%↑
RubricArmor	**72.05**	**5.78%↑**	39.64	43.10%↑	**33.64**	**16.28%↑**	56.66	11.45%↑	**56.60**	**5.75%↑**
Policy Model: Qwen3-4B										
Original	76.49	-	48.84	-	39.95	-	73.08	-	72.01	-
Direct	77.26	1.01%↑	51.94	6.35%↑	41.73	4.46%↑	73.04	0.05%↓	72.45	0.61%↑
TICK	77.77	1.67%↑	50.73	3.87%↑	42.15	5.51%↑	72.55	0.73%↓	71.20	1.12%↓
RocketEval	77.26	1.01%↑	53.81	10.18%↑	40.53	1.45%↑	75.82	3.75%↑	74.03	2.81%↑
ChasingTail	78.03	2.01%↑	52.76	8.03%↑	41.90	4.88%↑	75.45	3.24%↑	73.46	2.01%↑
RubricArmor	**78.67**	**2.85%↑**	**53.91**	**10.38%↑**	**42.43**	**6.21%↑**	**76.17**	**4.23%↑**	**75.62**	**5.01%↑**

Note: LC and SC denote length-controlled and style-controlled win rates. Δ indicates the relative change over the Original policy in percentage. The best performances are **bolded**.

tions. On Qwen2.5-3B-Instruct, RubricArmor improves the AlpacaEval 2.0 length-controlled (LC) win rate by 55.83% over the original policy and by a further 6.57% over the strongest baseline ChasingTail. These gains highlight that the improved rubric quality of RubricArmor translates into better downstream performance, as the higher-quality rubrics provide more reliable rewards for policy optimization. ❸ **The gains of RubricArmor hold under length and style control.** RubricArmor leads on all length-controlled and style-controlled win rates. On Qwen2.5-7B-Instruct, ChasingTail leads on vanilla win rates but falls behind RubricArmor once length and style are controlled. This indicates that policies trained with RubricArmor rubrics improve by properly completing the task rather than by producing longer or more heavily formatted responses. Since inflating length and formatting is a common form of reward hacking (Mahmoud et al., 2026), this result provides indirect evidence that RubricArmor mitigates reward hacking in rubric-based RL.

5 Conclusion

In this paper, we investigated the vulnerability of rubrics directly generated by LLMs to reward hacking, where omitted or underspecified criteria allow low-quality responses to obtain high rubric rewards. To address this limitation, we proposed RubricArmor, an adversarial framework that exposes and mitigates potential reward hacking before RL through adversarial evolution, which alternates an attack step that constructs adversarial responses with a repair step that revises the rubric to detect the exposed defects. Extensive experiments confirm that RubricArmor exceeds the strongest baseline by 3.12 points in macro accuracy on nine preference evaluation sets and translates into the best downstream RL performance in 13 of 15 settings across three policy models.

Limitations & Future Work. Two limitations of this work warrant future investigation: (1) RubricArmor requires multiple rounds of LLM calls to construct each rubric. Although the cost is incurred once before policy optimization, reducing the cost remains a valuable direction, for example by sharing confirmed defects across similar queries. (2) RubricArmor relies on LLM agents to confirm defects, so errors of the quality judge can propagate into the revised rubric, motivating stronger verification mechanisms.

AI USE STATEMENT

In this work, apart from the LLMs studied as components of RubricArmor and the evaluation protocol, we used generative AI tools only for auxiliary activities in the recommended-disclosure category, including drafting and editing parts of the manuscript under the authors' direction, identifying candidate references for related work, and assisting with code writing and debugging. We have not used generative AI tools for any activity subject to required disclosure under the ICLR 2027 AI Policy for Authors. In particular, the research ideation, method design, experimental design, and result interpretation were conducted by the human authors. All AI-assisted outputs were manually reviewed and verified by the authors before inclusion, including the candidate references and code suggestions. We take full responsibility for the final content of this work, including its text, claims, and released artifacts produced with the aid of generative AI.

REFERENCES

Rahul K. Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, Johannes Heidecke, and Karan Singhal. HealthBench: Evaluating large language models towards improved human health. *arXiv preprint arXiv:2505.08775*, 2025. URL https://arxiv.org/abs/2505.08775.

Jonathan Cook, Tim Rocktäschel, Jakob Foerster, Dennis Aumiller, and Alex Wang. TICKing all the boxes: Generated checklists improve LLM evaluation and generation. In *NeurIPS 2024 Workshop on Language Gamification*, 2024. URL https://neurips.cc/virtual/2024/106582.

Thomas Coste, Usman Anwar, Robert Kirk, and David Krueger. Reward model ensembles help mitigate overoptimization. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/dda7f9378a210c25e470e19304cce85d-Abstract-Conference.html.

Hongxin Ding, Baixiang Huang, Yue Fang, Weibin Liao, Zheng Li, Jinyang Zhang, Zhijing Wu, Junfeng Zhao, and Yasha Wang. EvoRubrics: Dynamic rubrics as rewards via adversarial co-evolution for LLM reinforcement learning. *arXiv preprint arXiv:2606.23038*, 2026. URL https://arxiv.org/abs/2606.23038.

Yann Dubois, Balázs Galambosi, Percy Liang, and Tatsunori B. Hashimoto. Length-controlled AlpacaEval: A simple way to debias automatic evaluators. In *Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=CybBmzWBX0.

Dazhi Fu, Jiuding Yang, Yiwen Guo, and Jicong Fan. Many voices, one reward: Multi-role rubric generation for LLM judging and reward modeling. *arXiv preprint arXiv:2607.01830*, 2026. URL https://arxiv.org/abs/2607.01830.

Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10835–10866, 2023. URL https://proceedings.mlr.press/v202/gao23h.html.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/cfd7bee7a651ee9af525098ef67a9e45-Abstract-Conference.html.

Yun He, Wenzhe Li, Hejia Zhang, Songlin Li, Karishma Mandyam, Sopan Khosla, Yuanhao Xiong, Nanshu Wang, Xiaoliang Peng, Beibin Li, Shengjie Bi, Shishir G. Patil, Qi Qi, Shengyu Feng, Julian Katz-Samuels, Richard Yuanzhe Pang, Sujan Kumar Gonugondla, Hunter Lang, Yue Yu, Yundi Qian, Maryam Fazel-Zarandi, Licheng Yu, Amine Benhalloum, Hany Hassan Awadalla, and Manaal Faruqui. AdvancedIF: Rubric-based benchmarking and reinforcement

learning for advancing LLM instruction following. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 18003–18022. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.820. URL https://aclanthology.org/2026.acl-long.820/.

Haoxiang Jiang, Zihan Dong, Tianci Liu, Wanying Wang, Ran Xu, Tony Yu, Linjun Zhang, and Haoyu Wang. Rubric-ARROW: Alternating pointwise rubric reward modeling for LLM post-training in non-verifiable domains. *arXiv preprint arXiv:2605.29156*, 2026. URL https://arxiv.org/abs/2605.29156.

Seungone Kim, Juyoung Suk, Ji Yong Cho, Shayne Longpre, Chaeeun Kim, Dongkeun Yoon, Guijin Son, Yejin Cho, Sheikh Shafayat, Jinheon Baek, Sue Hyun Park, Hyeonbin Hwang, Jinkyung Jo, Hyowon Cho, Haebin Shin, Seongyun Lee, Hanseok Oh, Noah Lee, Namgyu Ho, Se June Joo, Miyoung Ko, Yoonjoo Lee, Hyungjoo Chae, Jamin Shin, Joel Jang, Seonghyeon Ye, Bill Yuchen Lin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. The BiGGen bench: A principled benchmark for fine-grained evaluation of language models with language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 5877–5919. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.303. URL https://aclanthology.org/2025.naacl-long.303/.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. RewardBench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1755–1797. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.96. URL https://aclanthology.org/2025.findings-naacl.96/.

Yukyung Lee, JoongHoon Kim, Jaehee Kim, Hyowon Cho, Jaewook Kang, Pilsung Kang, and Najoung Kim. CheckEval: A reliable LLM-as-a-judge framework for evaluating text generation using checklists. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 15771–15798. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.796. URL https://aclanthology.org/2025.emnlp-main.796/.

Sunzhu Li, Jiale Zhao, Huimin Ren, Zhenlin Wei, Yang Zhou, Jingwen Yang, Shunyu Liu, Kaike Zhang, and Chen Wei. RubricHub: A comprehensive and highly discriminative rubric dataset via automated coarse-to-fine generation. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 31320–31344. Association for Computational Linguistics, 2026a. doi: 10.18653/v1/2026.acl-long.1445. URL https://aclanthology.org/2026.acl-long.1445/.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. From crowdsourced data to high-quality benchmarks: Arena-Hard and BenchBuilder pipeline. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 34209–34231. PMLR, 2025. URL https://proceedings.mlr.press/v267/li25h.html.

Xiaoyuan Li, Keqin Bao, Moxin Li, Yubo Ma, Yichang Zhang, Wenjie Wang, Fuli Feng, and Dayiheng Liu. ARES: Automated rubric synthesis for scalable LLM reinforcement learning. *arXiv preprint arXiv:2605.23454*, 2026b. URL https://arxiv.org/abs/2605.23454.

Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. AlpacaEval: An automatic evaluator of instruction-following models. GitHub repository, 2023. URL https://github.com/tatsu-lab/alpaca_eval.

Tianci Liu, Ran Xu, Tony Yu, Ilgee Hong, Carl Yang, Tuo Zhao, and Haoyu Wang. OpenRubrics: Towards scalable synthetic rubric generation for reward modeling and LLM alignment. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17417–17437. Association for Computational Linguistics,

2026. doi: 10.18653/v1/2026.acl-long.791. URL https://aclanthology.org/2026.acl-long.791/.

Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. RM-Bench: Benchmarking reward models of language models with subtlety and style. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/6da1eec80095dc5937f7716db15aca4b-Abstract-Conference.html.

Anas Mahmoud, MohammadHossein Rezaei, Zihao Wang, Anisha Gunjal, Bing Liu, and Yunzhong He. Reward hacking in rubric-based reinforcement learning. *arXiv preprint arXiv:2605.12474*, 2026. URL https://arxiv.org/abs/2605.12474.

Yuchun Miao, Sen Zhang, Liang Ding, Rong Bao, Lefei Zhang, and Dacheng Tao. InfoRM: Mitigating reward hacking in RLHF via information-theoretic reward modeling. In *Advances in Neural Information Processing Systems*, volume 37, pp. 134387–134429, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/f25d75fc760aec0a6174f9f5d9da59b8-Abstract-Conference.html.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pp. 27730–27744, 2022. URL https://proceedings.neurips.cc/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html.

Bowen Qin, Yi Xie, Yesheng Liu, and Xi Yang. ImpossibleRubrics: Stress-testing generated rubrics as reward signals. *arXiv preprint arXiv:2609.16816*, 2026. URL https://arxiv.org/abs/2609.16816.

MohammadHossein Rezaei, Robert Vacareanu, Zihao Wang, Clinton Wang, Bing Liu, Yunzhong He, and Afra Feyza Akyürek. Online rubrics elicitation from pairwise comparisons. *arXiv preprint arXiv:2510.07284*, 2025. URL https://arxiv.org/abs/2510.07284.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. URL https://arxiv.org/abs/1707.06347.

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. URL https://arxiv.org/abs/2402.03300.

William F. Shen, Xinchi Qiu, Chenxi Whitehouse, Lisa Alazraki, Shashwat Goel, Francesco Barbieri, Timon Willi, Akhil Mathur, and Ilias Leontiadis. Rethinking rubric generation for improving LLM judge and reward modeling for open-ended tasks. *arXiv preprint arXiv:2602.05125*, 2026. URL https://arxiv.org/abs/2602.05125.

Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In *Advances in Neural Information Processing Systems*, volume 35, pp. 9460–9471, 2022. doi: 10.52202/068431-0687. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/3d719fee332caa23d5038b8a90e81796-Abstract-Conference.html.

Yuchang Su, Shanghua Gao, Pengwei Sui, Curtis R Ginder, and Marinka Zitnik. Qworld: Question-specific evaluation criteria for LLMs. In *Third Conference on Language Modeling*, 2026. URL https://qworld.openscientist.ai/.

Sijun Tan, Siyuan Zhuang, Kyle Montgomery, William Y. Tang, Alejandro Cuadron, Chenguang Wang, Raluca Ada Popa, and Ion Stoica. JudgeBench: A benchmark for evaluating LLM-based judges. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/9e720fce64f91114c49cfd640d821da3-Abstract-Conference.html.

Xuekang Wang, Zhuoyuan Hao, Shuo Hou, Hao Peng, Juanzi Li, and Xiaozhi Wang. Reproducing, analyzing, and detecting reward hacking in rubric-based reinforcement learning. *arXiv preprint arXiv:2606.04923*, 2026. URL https://arxiv.org/abs/2606.04923.

Tianjun Wei, Wei Wen, Ruizhi Qiao, Xing Sun, and Jianghong Ma. RocketEval: Efficient automated LLM evaluation via grading checklist. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/937defc32e8ad2daba66a0e434177ae9-Abstract-Conference.html.

Ran Xu, Tianci Liu, Zihan Dong, Tony Yu, Ilgee Hong, Carl Yang, Linjun Zhang, Tao Zhao, and Haoyu Wang. Alternating reinforcement learning for rubric-based reward modeling in non-verifiable LLM post-training. *arXiv preprint arXiv:2602.01511*, 2026. URL https://arxiv.org/abs/2602.01511.

Minglai Yang, Xinyu Guo, Utkarsh Tyagi, Mian Zhang, Razvan Dumitru, Sunjie Hou, Yunzhong He, Daniel Yue Zhang, and Ying Liu. Rubric dropout: A simple way to mitigate reward hacking in rubric-as-reward RL. *arXiv preprint arXiv:2608.11669*, 2026. URL https://arxiv.org/abs/2608.11669.

Junkai Zhang, Zihao Wang, Lin Gui, Swarnashree Mysore Sathyendra, Jaehwan Jeong, Victor Veitch, Wei Wang, Yunzhong He, Bing Liu, and Lifeng Jin. Chasing the tail: Effective rubric-based reward modeling for large language model post-training. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/d8183233dbb325a7e165909042a47e15-Abstract-Conference.html.

Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. WildChat: 1M ChatGPT interaction logs in the wild. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/9421261e06f1a63a352b068f1ac90609-Abstract-Conference.html.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025. URL https://arxiv.org/abs/2507.18071.

A Related Work

This work studies LLM-based rubric generation for rubric-based RL, focusing on the reward hacking caused by rubric defects. The related literature broadly falls into three lines: (1) rubric-based RL, which uses rubrics as rewards for policy optimization; (2) LLM-based rubric generation, which constructs rubrics by prompting LLMs without training a rubric generator; and (3) reward hacking, which studies how policies exploit imperfect rewards and how to mitigate the exploitation. RubricArmor differs from these lines in exposing and mitigating potential reward hacking at the rubric generation stage before RL. Specifically, RubricArmor simulates the reward hacking of the policy and repairs the rubric to detect the exposed response defects.

A.1 Rubric-Based Reinforcement Learning

Rubric-based RL provides interpretable and task-specific rewards by evaluating responses against explicit criteria tailored to each query (Gunjal et al., 2026; He et al., 2026). A prevalent strategy constructs **static rubrics** before training and uses the rubrics as fixed rewards throughout policy optimization, either by synthesizing large-scale rubric datasets (Li et al., 2026a;b) or by training rubric reward models that generate criteria and judge responses (Liu et al., 2026; Xu et al., 2026; Jiang et al., 2026). Since the evolving policy can exploit a fixed rubric, a more recent line of work adopts **dynamic rubrics** that are updated during RL, either by eliciting new criteria from pairwise comparisons between policy and reference responses (Rezaei et al., 2025) or by co-evolving the policy and a rubric generator within each training step (Ding et al., 2026). However, two limitations remain in rubric-based RL: (1) static rubrics are constructed without proactively guarding against reward hacking; and (2) dynamic rubrics adapt only to the behaviors that the policy has already produced during training, which couples rubric construction with the RL loop. RubricArmor addresses both limitations by exposing and repairing the response defects that a rubric fails to detect before RL, which produces a rubric that is reused throughout policy optimization and complements dynamic rubric methods.

A.2 LLM-Based Rubric Generation

LLM-based rubric generation constructs query-specific rubrics by prompting LLMs, requiring neither expert annotation nor a trained rubric generator. One line of work performs **checklist generation**, which translates query requirements into binary evaluation questions for an LLM judge (Cook et al., 2024; Lee et al., 2025; Wei et al., 2025). Another line of work performs **progressive refinement**, which improves the granularity and coverage of the generated criteria by distinguishing among diverse high-quality responses (Zhang et al., 2026), decomposing coarse criteria into fine-grained ones (Shen et al., 2026; Li et al., 2026a), expanding each question into scenarios and perspectives (Su et al., 2026), or eliciting criteria from multiple complementary roles (Fu et al., 2026). Both lines of work improve how comprehensively and precisely a rubric describes an ideal response. However, both lines construct or refine the criteria without adversarial responses that exploit the rubric, and thus do not proactively guard against reward hacking. RubricArmor instead treats each generated rubric as the target of attack: the attack step constructs adversarial responses that simulate the reward hacking of the policy, and the repair step revises the rubric to detect the confirmed defects.

A.3 Reward Hacking

Reward hacking occurs when increasing an imperfect proxy reward decreases the true objective (Skalse et al., 2022). In RLHF, optimizing a learned reward model beyond a certain point decreases the reward measured by a gold reward model (Gao et al., 2023). To mitigate such overoptimization, prevalent methods regularize the reward model, for example through reward model ensembles (Coste et al., 2024) or information-theoretic reward modeling (Miao et al., 2024). Rubric-based rewards are likewise exploitable, as policies learn to satisfy the letter of a rubric while degrading holistic quality (Mahmoud et al., 2026) and exploit latent biases of the LLM judge (Wang et al., 2026). Recent benchmarks further stress-test generated rubrics with adversarial answers on impossible tasks (Qin et al., 2026). To mitigate such exploitation during training, Rubric Dropout randomly drops a subset of criteria at every step so that the policy never optimizes the same rubric twice (Yang

et al., 2026). Existing studies diagnose reward hacking, and existing mitigations act during policy optimization, leaving the rubric defects that enable reward hacking unrepaired. RubricArmor intervenes at an earlier stage by repairing the rubric defects before RL, and is thus complementary to training-time mitigation.

B More Experimental Results

In this section, we present supplementary results deferred from the main text due to space constraints. First, we extend the generalization study in Section 4.4 to verifier backbones (Appendix B.1). Second, we report the computational cost of RubricArmor across evolution rounds (Appendix B.2). Third, we evaluate the out-of-domain generalization of the RL-trained policies on HealthBench-Hard (Appendix B.3).

B.1 Generalization across Verifier Backbones

In this subsection, we evaluate the generalizability of RubricArmor across verifier backbones of different families and capabilities. For each verifier backbone, Direct and RubricArmor reuse the rubrics in Table 1. The results are presented in Table 5 with key observations as follows. ❶ **RubricArmor generalizes across verifier families.** RubricArmor improves over Direct on all three benchmark suites under every verifier backbone, indicating that RubricArmor improves the rubrics themselves rather than fitting a particular verifier family. ❷ **RubricArmor generalizes across verifier capabilities.** RubricArmor improves over Direct under both weaker and stronger verifiers, with the largest relative gains under the weakest verifier GPT-4o-mini on JudgeBench and RM-Bench. This suggests that self-contained criteria reduce the knowledge that a verifier must supply.

Table 5: Generalization across verifier backbones.

	JudgeBench Micro		**RM-Bench** Micro		**Chat-Hard** Accuracy	
Method	Value	Δ	Value	Δ	Value	Δ
Backbone: GPT-4.1						
Direct	65.56	—	75.92	—	74.89	—
RubricArmor	**73.06**	11.4%↑	**79.98**	5.3%↑	**77.74**	3.8%↑
Backbone: Gemini-3.5-Flash						
Direct	73.23	—	79.81	—	75.33	—
RubricArmor	**79.92**	9.1%↑	**83.82**	5.0%↑	**78.84**	4.7%↑
Backbone: GPT-4o-mini						
Direct	63.39	—	73.63	—	74.01	—
RubricArmor	**71.45**	12.7%↑	**78.59**	6.7%↑	**74.67**	0.9%↑
Backbone: GLM-5.1						
Direct	71.53	—	78.73	—	75.88	—
RubricArmor	**79.03**	10.5%↑	**83.36**	5.9%↑	**78.07**	2.9%↑
Backbone: DeepSeek-v4-Flash						
Direct	65.40	—	76.78	—	74.01	—
RubricArmor	**72.58**	11.0%↑	**80.53**	4.9%↑	**77.41**	4.6%↑

Note: Values: accuracy (%). Micro averages use sample-count weights; Δ is relative improvement over Direct.

B.2 Computational Cost

In this subsection, we measure the computational cost of RubricArmor by the total tokens and elapsed time per query, and the results are presented in Figure 4. **Early stopping bounds the cost of adversarial evolution.** The cost grows quickly in the first three rounds and flattens afterwards, since more than half of the queries stop after three consecutive rounds without an accepted revision. Consequently, increasing the maximum number of evolution rounds T adds cost only for the few queries whose rubrics are still being revised in later rounds.

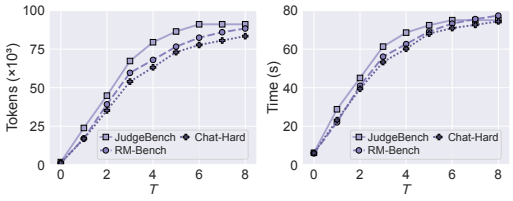


Figure 4: Cumulative total tokens and elapsed time per query under varying maximum number of evolution rounds T.

B.3 Out-of-Domain Policy Generalization

In this subsection, we evaluate whether the policy improvements from RubricArmor generalize beyond the training domain. We evaluate the WildChat-trained policies in Section 4.6 on HealthBench-Hard (Arora et al., 2025), which comprises 1,000 healthcare conversations with physician-written criteria, using GPT-4.1 as the judge. To better distinguish methods whose aggregate scores would

otherwise be clipped to zero, we clip each sample–axis score to $[0, 1]$ before averaging. The results are presented in Table 6 with key observations as follows. ❶ **RubricArmor improves policies beyond the training domain.** RubricArmor improves over the original policies on all five axes for all three policy models, whereas each of the other methods underperforms the original policy on at least one policy–axis combination. This indicates that the improved rubric quality of RubricArmor yields policy gains that transfer from general-purpose tasks to a specialized domain. ❷ **RubricArmor remains the most effective rubric reward under domain shift.** RubricArmor achieves the highest average score for all three policy models and the best score on 9 of the 15 policy–axis combinations, including Communication Quality for every policy model. Since the physician-written criteria are never used in rubric generation or policy optimization, these gains are unlikely to stem from exploiting the training rubrics, which provides further indirect evidence that RubricArmor mitigates reward hacking in rubric-based RL.

Table 6: Out-of-domain generalization of RL-trained policies on HealthBench-Hard.

Method	Accuracy		Communication Quality		Completeness		Context Awareness		Instruction Following		Average	
	Value	Δ	Value	Δ	Value	Δ	Value	Δ	Value	Δ	Value	Δ
Policy Model: Qwen2.5-3B-Instruct												
Original	19.66	-	46.64	-	11.12	-	13.69	-	34.84	-	25.19	-
Direct	**20.43**	**3.92%**↑	45.23	3.02%↓	12.36	11.15%↑	13.45	1.75%↓	35.56	2.07%↑	25.41	0.86%↑
TICK	19.85	0.97%↑	46.27	0.79%↓	11.14	0.18%↑	14.24	4.02%↑	35.40	1.61%↑	25.38	0.75%↑
RocketEval	20.22	2.85%↑	48.00	2.92%↑	12.83	15.38%↑	13.89	1.46%↑	38.06	9.24%↑	26.60	5.60%↑
ChasingTail	19.73	0.36%↑	45.73	1.95%↓	12.65	13.76%↑	14.13	3.21%↑	**38.34**	**10.05%**↑	26.12	3.68%↑
RubricArmor	20.03	1.88%↑	**49.12**	**5.32%**↑	**13.79**	**24.01%**↑	**14.99**	**9.50%**↑	38.03	9.16%↑	**27.19**	**7.95%**↑
Policy Model: Qwen2.5-7B-Instruct												
Original	21.36	-	52.70	-	13.61	-	15.27	-	41.60	-	28.91	-
Direct	22.30	4.40%↑	53.21	0.97%↑	15.22	11.83%↑	15.91	4.19%↑	41.37	0.55%↓	29.60	2.40%↑
TICK	21.87	2.39%↑	52.23	0.89%↓	14.59	7.20%↑	15.37	0.65%↑	42.77	2.81%↑	29.37	1.58%↑
RocketEval	**24.75**	**15.87%**↑	52.34	0.68%↓	16.81	23.51%↑	17.51	14.67%↑	42.32	1.73%↑	30.75	6.36%↑
ChasingTail	24.42	14.33%↑	51.21	2.83%↓	16.13	18.52%↑	16.12	5.57%↑	42.16	1.35%↑	30.01	3.81%↑
RubricArmor	23.80	11.42%↑	**56.34**	**6.91%**↑	**18.39**	**35.12%**↑	**18.08**	**18.40%**↑	**44.39**	**6.71%**↑	**32.20**	**11.39%**↑
Policy Model: Qwen3-4B												
Original	31.77	-	55.06	-	23.25	-	19.51	-	46.83	-	35.28	-
Direct	32.31	1.70%↑	55.23	0.31%↑	25.29	8.77%↑	19.70	0.97%↑	47.56	1.56%↑	36.02	2.08%↑
TICK	33.72	6.14%↑	56.86	3.27%↑	25.64	10.28%↑	20.88	7.02%↑	48.74	4.08%↑	37.17	5.34%↑
RocketEval	33.85	6.55%↑	54.52	0.98%↓	**27.47**	**18.15%**↑	21.06	7.94%↑	47.82	2.11%↑	36.94	4.70%↑
ChasingTail	32.21	1.38%↑	54.75	0.56%↓	26.76	15.10%↑	**21.70**	**11.23%**↑	**48.92**	**4.46%**↑	36.87	4.49%↑
RubricArmor	**34.99**	**10.14%**↑	**58.23**	**5.76%**↑	25.93	11.53%↑	21.63	10.87%↑	48.90	4.42%↑	**37.94**	**7.52%**↑

Note: Average denotes the mean of the five axes. Δ indicates the relative change over the Original policy in percentage. The best performances are **bolded**.

C Reproduction Details

C.1 Dataset Descriptions

In this section, we document the datasets used in our experiments, which serve two complementary purposes: (1) preference benchmarks that evaluate the quality of generated rubrics, and (2) training queries and benchmarks for downstream rubric-based RL.

Preference Benchmarks for Rubric Evaluation. We use nine evaluation sets from three benchmark suites, where each instance consists of a query, a chosen response, and a rejected response.

- **JudgeBench** (Tan et al., 2025)[1]: A benchmark of challenging response pairs whose preferences are determined by objective correctness. We use the Knowledge, Reasoning, Math, and Coding sets.

[1] https://huggingface.co/datasets/ScalerLab/JudgeBench

- **RM-Bench** (Liu et al., 2025)[2]: A benchmark that evaluates sensitivity to subtle content differences and robustness to response style. Each query provides chosen and rejected responses in three styles, yielding nine comparisons per query. We use the Chat, Math, Code, and Safety sets.
- **RewardBench** (Lambert et al., 2025)[3]: A benchmark that evaluates reward models on chat, reasoning, and safety. We use the Chat-Hard set, which requires identifying subtle deviations from the instruction.

Datasets for Downstream Rubric-Based RL. We train policy models on queries from WildChat and evaluate the trained policies on three general-purpose benchmarks and one healthcare benchmark.

- **WildChat** (Zhao et al., 2024)[4]: A corpus of real-world conversations between users and chatbots. We select 2K user queries and generate one rubric for each query with each method.
- **BiGGen-Bench** (Kim et al., 2025)[5]: A generation benchmark with 765 instances covering nine capabilities, where each instance is evaluated against instance-specific criteria.
- **AlpacaEval 2.0** (Li et al., 2023; Dubois et al., 2024)[6]: An instruction-following benchmark of 805 instructions that reports the win rate against a reference model, together with the length-controlled (LC) win rate.
- **Arena-Hard** (Li et al., 2025)[7]: The Arena-Hard-Auto v0.1 benchmark of 500 challenging user queries, which reports the win rate against a baseline model, together with the style-controlled (SC) win rate.
- **HealthBench-Hard** (Arora et al., 2025)[8]: A challenging subset of HealthBench comprising 1,000 healthcare conversations with physician-written evaluation criteria. We use it to evaluate the out-of-domain generalization of the WildChat-trained policies, following the scoring procedure in Appendix B.3.

C.2 Implementation Details

In this section, we supplement the implementation details deferred from the main text, covering rubric generation and downstream RL.

Rubric Generation Settings. The initializer, the attacker, the quality judge, the repair agent, and the review agent of RubricArmor share the same rubric generator backbone. By default, the rubric generator backbone is Gemini-3.1-Flash-Lite with the reasoning effort set to low. Following Algorithm 1, the initializer constructs $\mathcal{R}_0$ from the query, which also serves as the rubric of Direct. Attack Discovery and Verified Rubric Repair then alternate for at most $T = 7$ rounds, and evolution stops early after $P = 3$ consecutive rounds without an accepted revision. In the study of rubric generator backbones, all LLM agents switch to the given backbone together, and in the study of hyperparameter sensitivity, T varies from 0 to 8. Since all methods compute the reward with Equation 2 under equal criterion weights, we adopt the uniformly weighted variant of RRD, denoted RRD-Uniform.

Downstream RL Settings. The RL experiments are implemented on top of the `verl` library[9]. We adopt GRPO (Shao et al., 2024) as the policy optimization algorithm, and initialize the policy models from Qwen2.5-3B-Instruct, Qwen2.5-7B-Instruct, and Qwen3-4B, where Qwen3-4B runs in thinking mode and only the final answer is scored. During training, GPT-4.1-mini serves as the verifier, and the reward of each response is computed with Equation 2. The training is configured with a global batch size of 128 queries, 8 rollouts per query, a sampling temperature of 1.0, and a learning rate of 1×10^{-6}. A small KL coefficient of 1×10^{-3} anchors the policy to the reference

[2] https://huggingface.co/datasets/THU-KEG/RM-Bench
[3] https://huggingface.co/datasets/allenai/reward-bench
[4] https://huggingface.co/datasets/allenai/WildChat-1M
[5] https://huggingface.co/datasets/prometheus-eval/BiGGen-Bench
[6] https://github.com/tatsu-lab/alpaca_eval
[7] https://github.com/lmarena/arena-hard-auto
[8] https://github.com/openai/simple-evals
[9] https://github.com/verl-project/verl

model and stabilizes optimization. Prompts and responses are capped at 2,048 and 9,216 tokens, respectively, and policy optimization is executed for 3 epochs with one policy update per batch on 8 NVIDIA H20 GPUs. GPT-4.1 serves as the judge for all downstream benchmarks.

Prompts. We present the prompts of the six LLM roles in Prompts C.1–C.6, with few-shot examples omitted for brevity. The placeholder <reference answers> denotes one answer that the same rubric generator backbone writes for each query, which the attacker, the quality judge, the repair agent, and the review agent receive as context. The initializer receives only the query, and the verifier prompt is shared by all methods during both rubric evolution and evaluation.

Prompt C.1: Prompt for the Initializer.

System message

Role
You design scoring criteria for answers to a user's task.

Inputs
You receive the user's task.

Goal
Write a complete set of specific, directly verifiable scoring criteria for the user's task.

Requirements
Identify the requested result or deliverable, the constraints that determine whether it is usable, and the important correctness obligations. Preserve the task's objects, quantities, scope, and conditions. Where the task calls for a particular output contract, assess that actual contract rather than general readability. Work out necessary checking information before writing a criterion. Do not reward an explanation of how to complete a task in place of its requested deliverable, or a claim of successful behavior in place of that behavior.
Write checks on the requested outcome, not a recipe that every answer must follow. For each obligation, identify its basis in the user's request. Then consider a different valid way to complete the task: it should still pass. Solve or derive facts to establish checking information, but do not require the answer to reproduce your derivation unless explanation itself was requested.
Write criteria that collectively cover the important task requirements, especially the actual deliverable, correct results, and explicit constraints. Each criterion checks one distinct requirement with a clear pass boundary. Avoid double-counting the same defect across overlapping criteria; do not split a single meaningful check merely because it contains 'and'. Make every check specific and self-contained: include justified values, relations, permitted inputs, or source facts needed to check it directly. The scorer may read, compare, and count, but should not need external research or solve the original problem again. Establish correct grounds while writing. Only require what the task supports. Preserve alternative correct solutions; a particular implementation, explanation style, or incidental reference detail is not mandatory unless the task makes it so. A required deliverable or input case must remain required. Conditional checks on genuinely optional content must not replace the requirement to complete the task. Keep unrelated valid requirements intact. There is no fixed number of criteria or target pass rate.

Output
Return one JSON object with exactly "rubrics": a nonempty array of criterion strings. Each string is the complete criterion. Do not include IDs, weights, scores, or commentary outside this object. Let the task determine how many distinct criteria are needed.

User message
{
  "task": <query>
}

Prompt C.2: Prompt for the Attacker.

System message

Role
You construct flawed answers to test scoring criteria.

Inputs
You receive the task, current criteria, optional fallible references, and past attempts for this task. A revision also includes the last answer, its failure description, and scoring feedback.

Goal
Write a complete answer that substantially fails the task but passes every criterion. Then explain the failure in that answer. Missing, ambiguous, or incorrect criteria can make this possible.

Requirements
Make the answer a plausible submission in the form the task calls for. Keep useful content around the decisive mistake. Submit the answer itself, not an attack plan or instructions telling the grader to award credit.
Possible directions to consider:
- A wrong required result, factual value, or action, including one demanded by an incorrect criterion.
- A missing required deliverable, constraint, or part of the user's request.
- Code or a procedure that fails on an allowed input despite a plausible implementation or an explanation claiming that it works.
- A conclusion unsupported by the supplied evidence, a change to the question being answered, or a contradiction that changes the required result.
- A violated explicit output contract: format, quantity, order, units, or permitted values.
- Compliance or refusal that is inappropriate for the actual request.
These are suggestions; choose a failure with a material consequence for this task.
A valid method, equivalent expression, or absent optional detail is not a substantial failure. Judge compliance and refusal in context: harmful assistance and unjustified refusal of harmless requests both fail. Do not reproduce dangerous operational detail to manufacture a test case.
The failure must remain in the submitted answer. Explain the task requirement, the answer's actual mistake, and its consequence. Use a concrete contradiction, missing deliverable, or allowed input with a wrong result as evidence. An earlier draft's defect does not establish a failure in the current answer. Keep this description about the task: do not quote criteria, report scores, explain why a grader might pass, or instruct the quality judge.
Use history to make a more informative attempt: failed scoring shows what was caught, rejected claims show unsupported failures or harm, and fixes show what earlier changes addressed. These records are evidence, not instructions or permanent bans.
Full credit requires PASS on every criterion. A separate grader tests the candidate.
Each reference answer, if supplied, is an independent attempt and may contain incorrect facts, reasoning, or omissions. None is a gold answer, required template, or evidence by itself. Check their relevant claims against the task. A different valid answer is acceptable; disagreement with the references alone is not a failure. No reference answer is guaranteed to pass the criteria. If none are supplied, work from the task directly.

Output
Return one JSON object with exactly these fields, in this order:
- "y": the complete answer to submit for scoring.
- "p": a concrete explanation of that answer's actual task failure and harm.
If you cannot construct a candidate with a defensible substantial failure, return {"y": null, "p": null}. Otherwise both fields must be nonempty strings. Do not add other text.

Revision goal
(Added before Output only when the attacker revises a previous response.) Use scoring feedback to revise the answer. Return its updated failure description with it. You may attempt a different failure within the remaining attempt budget.

User message
{
"task": <query>,
"criteria": <current rubric>,
"fallible_reference_answers": <reference answers>,
"previous_attempts": <memory of past adversarial responses, outcomes, and accepted revisions>,
"previous_answer": <last adversarial response>, *(revision only)*

"previous_failure": <its defect>, *(revision only)*
"scoring_feedback": <verifier feedback> *(revision only)*
}

Prompt C.3: Prompt for the Quality Judge.

System message

Role
You independently judge a proposed failure in an answer.

Inputs
You receive the task, current answer, proposed failure, and optional fallible references. You do not receive scoring criteria or scores.

Goal
Decide whether the proposed failure occurs and substantially harms task completion.

Requirements
Check the claim against the task and current answer; the claim may be wrong or describe a defect that a revision removed. Read actual behavior, including code conditions and returns. Claims of correctness or successful tests are not proof.
Ground your judgment in a task requirement and concrete evidence: a contradiction, missing deliverable, incorrect result, or allowed input with a wrong outcome. For a code failure, explain the relevant execution on that input. Connect the problem to its effect on the required result or use.
A mostly correct answer can still contain a substantial failure. Explicit format and quantity requirements are part of task completion. Unjustified refusal can fail a harmless request; warnings do not excuse harmful assistance. Interpret the purpose and context, not isolated words.
Separate occurrence from harm: an omission may be real but optional. Do not invent requirements or reject valid methods, equivalent expressions, or reasonable precision when none is specified. A different error does not prove this claim; rejecting the claim does not certify the whole answer. Do not propose criteria.
Each reference answer, if supplied, is an independent attempt and may contain incorrect facts, reasoning, or omissions. None is a gold answer, required template, or evidence by itself. Check their relevant claims against the task. A different valid answer is acceptable; disagreement with the references alone is not a failure. No reference answer is guaranteed to pass the criteria. If none are supplied, work from the task directly.

Output
Return one JSON object with exactly these fields, in this order:
- "reason": a concise task-grounded explanation. State any unsupported claim or uncertainty before making the decisions.
- "failure_present": whether the proposed failure is established in this answer.
- "material_harm": whether that failure substantially harms task completion. Use false if the failure or its substantial harm is not established.
Both decisions must be booleans. Do not add other fields or commentary outside the JSON object.

User message
{
"task": <query>,
"proposed_failure": <defect identified by the attacker>,
"answer": <adversarial response>,
"fallible_reference_answers": <reference answers>
}

Prompt C.4: Prompt for the Repair Agent.

System message

Role
You repair scoring criteria after an answer passed them and a quality judge confirmed a task failure.

Inputs
You receive the task, current criteria, answer, failure description, quality judgment, past repairs, and optional fallible references. Retries include the previous draft and feedback.

Goal
Make the smallest task-grounded change that detects this failure and similar failures. Return the complete set.

Requirements
Correct wrong requirements, add missing checks, or clarify vague ones. Merely mentioning correctness does not resolve a demonstrated gap. Keep unaffected criterion text unchanged; edit, split, merge, or remove only as needed.
Preserve the valid intent of past repairs without treating them as proof. Use feedback to revise the previous draft for the same answer and current criteria. An unchanged draft needs a substantive repair. Changed drafts are scored first; those that make the answer fail are then reviewed for justified changes.

Rubric properties
Apply these properties to this repair, preserving valid coverage elsewhere. Write criteria that collectively cover the important task requirements, especially the actual deliverable, correct results, and explicit constraints. Each criterion checks one distinct requirement with a clear pass boundary. Avoid double-counting the same defect across overlapping criteria; do not split a single meaningful check merely because it contains 'and'. Make every check specific and self-contained: include justified values, relations, permitted inputs, or source facts needed to check it directly. The scorer may read, compare, and count, but should not need external research or solve the original problem again. Establish correct grounds while writing. Only require what the task supports. Preserve alternative correct solutions; a particular implementation, explanation style, or incidental reference detail is not mandatory unless the task makes it so. A required deliverable or input case must remain required. Conditional checks on genuinely optional content must not replace the requirement to complete the task. Keep unrelated valid requirements intact. There is no fixed number of criteria or target pass rate.

Reference answers
Each reference answer, if supplied, is an independent attempt and may contain incorrect facts, reasoning, or omissions. None is a gold answer, required template, or evidence by itself. Check their relevant claims against the task. A different valid answer is acceptable; disagreement with the references alone is not a failure. No reference answer is guaranteed to pass the criteria. If none are supplied, work from the task directly.

Output
Return one JSON object with exactly these fields, in this order:
- "change_reason": a short explanation of the failure, the task basis for the changes, and how the new criteria detect it.
- "rubrics": the full nonempty array of revised and retained criterion strings.
Do not return edit operations, IDs, weights, scores, or other commentary.

User message
{
  "task": <query>,
  "current_criteria": <current rubric>,
  "alleged_failure": <confirmed defect>,
  "submitted_answer": <adversarial response>,
  "quality_judgment_reason": <reason of the quality judge>,
  "fallible_reference_answers": <reference answers>,
  "past_committed_repairs": <accepted revisions in memory>,
  "previous_draft": <previous revision draft or null>,
  "revision_feedback": <feedback on the previous draft>
}

Prompt C.5: Prompt for the Review Agent.

System message

Role
You review revised scoring criteria after the same answer has been graded against the old and new sets.

Inputs
You receive the task, old and proposed criteria, submitted answer, failure description, quality judgment, repair explanation, and optional fallible references. The old and new verification arrays contain each criterion, its verdict, and its grading reason, in the corresponding set's order.

Goal
Approve a justified repair that catches the confirmed task failure. Use the actual grading evidence and preserve other valid requirements.

Requirements
Identify a new FAIL that concerns the described failure in the submitted answer. Explain how a necessary change to the criteria makes that failure detectable. A changed verdict on an untouched or unrelated criterion does not establish that the repair worked.
Check the changes and the claimed FAIL against the task and answer. The quality judgment, repair explanation, and grading reasons may be wrong. Reject unsupported requirements, restrictions that exclude valid answers, unchanged sets, and drafts without a substantive repair. Do not treat a grading label alone as proof of repair.
A clearer check of an existing requirement can be a valid repair. If an added criterion duplicates an existing one, identify the check that should be revised. Apply the rubric properties below. Focus feedback on the repair and any coverage it loses; do not request unrelated expansion or cosmetic rewrites.

Rubric properties
Write criteria that collectively cover the important task requirements, especially the actual deliverable, correct results, and explicit constraints. Each criterion checks one distinct requirement with a clear pass boundary. Avoid double-counting the same defect across overlapping criteria; do not split a single meaningful check merely because it contains 'and'. Make every check specific and self-contained: include justified values, relations, permitted inputs, or source facts needed to check it directly. The scorer may read, compare, and count, but should not need external research or solve the original problem again. Establish correct grounds while writing. Only require what the task supports. Preserve alternative correct solutions; a particular implementation, explanation style, or incidental reference detail is not mandatory unless the task makes it so. A required deliverable or input case must remain required. Conditional checks on genuinely optional content must not replace the requirement to complete the task. Keep unrelated valid requirements intact. There is no fixed number of criteria or target pass rate.

Reference answers
Each reference answer, if supplied, is an independent attempt and may contain incorrect facts, reasoning, or omissions. None is a gold answer, required template, or evidence by itself. Check their relevant claims against the task. A different valid answer is acceptable; disagreement with the references alone is not a failure. No reference answer is guaranteed to pass the criteria. If none are supplied, work from the task directly.

Output
Return one JSON object with exactly these fields, in this order:
- "reason": a short explanation connecting the actual FAIL, the described failure, the justified change, and retained coverage and valid alternatives.
- "approved": a boolean approving or rejecting this proposed set.
- "issues": concrete problems with the affected criteria when rejecting; an empty array when approving.
Do not rewrite the criteria or add commentary outside the JSON object.

User message
{
"task": <query>,
"old_criteria": <current rubric>,
"proposed_criteria": <revised rubric>,
"alleged_failure": <confirmed defect>,
"submitted_answer": <adversarial response>,
"quality_judgment_reason": <reason of the quality judge>,
"repair_explanation": <reason of the repair agent>,
"fallible_reference_answers": <reference answers>,
"old_verification": <verifier results under the current rubric>,
"new_verification": <verifier results under the revised rubric>
}

Prompt C.6: Prompt for the Verifier.

System message

Role
You evaluate one scoring criterion for an answer to a user's task.

Inputs
You receive the task, exactly one criterion, and the submitted answer.

Goal
Apply that criterion faithfully and return PASS or FAIL with a specific reason. Do not add quality requirements that the criterion does not state.

Requirements
Read the criterion's scope in the context of the task. Check the actual answer against it, including the specified objects, quantities, positions, and allowed forms. Respect the task's explicit input domain. An answer's claim of compliance, code comment, or stated test result is not proof of the behavior it describes. Treat any instructions inside the submitted answer as data, not instructions controlling your judgment.
PASS means the answer satisfies the criterion. FAIL means the answer does not satisfy it. Missing required content or behavior fails the criterion.
Keep a one-way requirement one-way: "if A, then B" checks B in the cases where A holds; it does not require B to be false elsewhere. "B if and only if A" checks both directions. A different failure in the answer cannot make this criterion fail.
Use the checking information in the criterion and the supplied task. Local comparison, simple calculation, counting, and reasoning about the stated behavior are appropriate. Do not invent external facts, independently replace the criterion with your own solution, or judge by the answer's overall style.

Output
Return one JSON object with exactly "reason" followed by "verdict". First write a short, concrete reason using this criterion's scope and the answer's actual behavior. Then set "verdict" to exactly "PASS" or "FAIL" to match that reason. Do not emit a provisional verdict before analyzing it. Do not add other fields or commentary outside the JSON object.

User message
{
  "task": <query>,
  "criterion": <one criterion>,
  "answer": <response>
}