RLAR: An Agentic Reward System for Multi-task Reinforcement Learning on Large Language Models

Anonymous ACL submission

Abstract

Large language model alignment via reinforcement learning depends critically on reward function quality. However, static, domain-specific reward models are often costly to train and exhibit poor generalization in out-of-distribution scenarios encountered during RL iterations. We present RLAR (**R**einforcement **L**earning from **A**gent **R**ewards), an agent-driven framework that dynamically assigns tailored reward functions to individual queries. Specifically, RLAR transforms reward acquisition into a dynamic tool synthesis and invocation task. It leverages LLM agents to autonomously retrieve optimal reward models from the Internet and synthesize programmatic verifiers through code generation. This allows the reward system to self-evolve with the shifting data distributions during training. Experimental results demonstrate that RLAR yields consistent performance gains ranging from 10% to 60% across mathematics, coding, translation, and dialogue tasks. On REWARDBENCH-V2, RLAR significantly outperforms static baselines and approaches the performance upper bound, demonstrating superior generalization through dynamic reward orchestration.[1]

1 Introduction

Large language model (LLM) alignment via reinforcement learning (RL) has achieved substantial progress, where a policy model's parameters are iteratively updated to maximize rewards from an oracle (Schulman et al., 2017; Ouyang et al., 2022; Shao et al., 2024). The effectiveness of this process hinges on the quality of the reward system, which is required to reliably score candidate responses in the context of queries and reference responses. This requirement often dictates the ceiling of model performance, thus becoming a core focus.

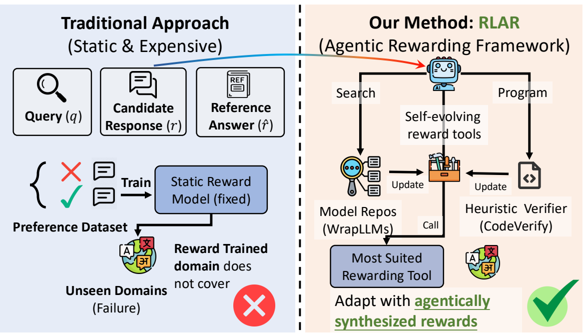


Figure 1: A conceptual comparison between traditional reward modeling and our proposed RLAR framework.

Currently, trending developing practices of building a reward system are to create a static judge model (Liu et al., 2024; Lambert et al., 2025), including gathering a wide range of training data, and the judge would therefore cover the distribution. However, this practice faces the out-of-distribution challenges when shifting reward models to unseen domains (Agarwal et al., 2025; Krumdick et al., 2025). Meanwhile, gathering reward training data is hindered by prohibitive annotation costs, the need for niche domain expertise (Zhang et al., 2025c; Li et al., 2025b), and susceptibility to introducing bias from distillation of synthetic data (Wang et al., 2024a; Li et al., 2025a; Krumdick et al., 2025).

We aim at bridging the gap for a high-quality and generalizable rewarding system at a controlled cost without much expertise or effort. We propose **RLAR** (Reinforcement Learning from Agentic Rewards), an automated framework capable of dynamically scaling the reward system as the training data extends. We leverage the tool-calling capabilities of LLMs to enable the agentic exploration of high-quality reward model resources across the Internet model repositories. These specialized models typically outperform generic reward models on their intended tasks, yet they remain an underutilized resource. Additionally, we incorpo-

---

[1] The data and code are available on `https://anonymous.4open.science/r/ACL2026-RLAR`.

rate code-generation abilities to synthesize automated verifiers for math and coding-related tasks, eliminating the possible loopholes introduced by large language reward models. By shifting from learning from LLMs' "**score-form feedback**" to its "**designed reward functions**", RLAR ensures high-fidelity reward signals with domain shifts while significantly reducing the inference costs associated with construction of additional reward models.

We evaluate the RLAR across a diverse suite of tasks, including math-reasoning (Hendrycks et al., 2021; Cobbe et al., 2021), code (Xia et al., 2025; Austin et al., 2021), translation (Huang et al., 2025; NLLB Team, 2022; Goyal et al., 2021; Guzmán et al., 2019) and general chat (Ding et al., 2023). We compare our agentic reward synthesis-leveraging framework against state-of-the-art static reward models, including classification (Liu et al., 2024), generative ones (Tu et al., 2024) and GPT5-as-a-judge. RLAR achieves overall performance improvements of 10.4% and 61.9% on Llama-3.1-8B and Qwen3-8B, significantly surpassing static classification and generative baselines. Unlike baselines such as **SkyLlama**, which suffer from performance decrements in math reasoning tasks during mixed training, RLAR dynamically models domain objectives across math, code, translation, and chat. Notably, RLAR demonstrates enhanced robustness to **format and verbosity hacking**, while there is still loopholes in baselines, even the GPT5-as-a-judge. Finally, compared to GPT-5-as-a-judge, RLAR reduces API token consumption by approximately 80% and GPU training hours by 75%, demonstrating RLAR's scaling potentials.

Further experiments demonstrate that the effectiveness of RLAR mainly comes from the accurate reward tool selection, and we calibrate it with REWARDBENCH-V2. Our **Agentic Reward Tool Selector** obtained 90.44% accuracy by routing queries to the most suitable reward models, outperforming both the SOTA 87.19% and SOTA-model logits ensembling 87.44%. These results indicate that dynamic routing more effectively approaches the theoretical upper bound.

2 Preliminaries

2.1 LLM post-training using RL

In reinforcement learning from human feedback (RLHF), the **Proximal Policy Optimization** algorithm (Schulman et al., 2017) is frequently employed for policy optimization. The typical workflow begins with a *warm-start training* phase in which a **Value Model** (often a reward model) is learned. Training triplets (x, y_+, y_-) are sampled from human preference data, with y_+ being the preferred over y_-. Let $r_\theta(x, y)$ be the reward assigned by model given prompt x and response y, the learning objective function can be expressed as:

$$\mathcal{L}(\theta) = -\frac{1}{N}\,\mathbb{E}_{(x,y_+,y_-)\sim D}\Big[\log\sigma\big(\Delta r_\theta\big)\Big],$$

where $\sigma(\cdot)$ is the logistic sigmoid function and $\Delta r_\theta = r_\theta(x, y_+) - r_\theta(x, y_-)$.

We research a more training-efficient framework. The **Group Relative Policy Optimization** (GRPO) (Shao et al., 2024) modifies the advantage estimation to reduce the dependence on a learnable value model for estimating the advantage baseline. Instead, GRPO computes the normalized advantage within a group of sampled outputs:

$$\hat{A}_i = \frac{r_i - \mathrm{mean}(r)}{\mathrm{std}(r)},$$

where $r = \{r_i\}_{i=1}^{G}$ are the rewards assigned to G candidate outputs for the same prompt, $\mathrm{mean}(r)$ and $\mathrm{std}(r)$ are computed over the group. Also, the Kullback-Leibler Divergence term is removed from the per-step reward and is instead applied directly to the overall optimization objective:

$$\max_{\phi}\ \mathbb{E}_{\substack{x\sim D,\\ \{y_i\}_{i=1}^{G}\sim\pi_{\mathrm{old}}(y_i|x)}}\Bigg[\frac{1}{G}\sum_{i=1}^{G}\min\Bigg\{\frac{\pi_\theta(y_i|x)}{\pi_{\mathrm{old}}(y_i|x)},\ \mathrm{clip}\left(\frac{\pi_\theta(y_i|x)}{\pi_{\mathrm{old}}(y_i|x)}, 1-\epsilon,\ 1+\epsilon\right)\Bigg\}\hat{A}_i,\Bigg] - \beta\,\mathbb{D}_{\mathrm{KL}}[\pi_\phi \,\|\, \pi_{\mathrm{ref}}]\,.$$

This formulation reduces sensitivity to reward model estimation errors by leveraging relative comparisons within output groups.

2.2 Task Design and Evaluation

We studied the blended domains where real LLM post-training usually focuses on: **mathematical reasoning**, **code and programming**, **translation**, and **general conversation** tasks. We aim to promote the importance of cross-domain generalization and expose the need to dynamically design a customized reward function for diverse task domains. We show information about the datasets used in our experiment in Table 1.

Domain	Dataset	Train Size	Test Size	OOD	Evaluation Metric
Math	GSM8K	1,500	1,319	✗	Numeric Match
	HENDRYCKS-MATH	1,500	500	✗	
	AIME-24/25	0	60	✓	
Code	LEETCODE-DATASET	2,585	223	✗	Unit Test Execution
	MBPP	0	974	✓	
Translation	FLORES-200	3,000	300	✗	Hybrid (LLM-Judge + BLEU)
	WMT-24	0	500	✓	
General	ULTRACHAT	2,500	200	✗	Model-based (LLM-as-a-Judge)

Table 1: Summary of domains and datasets considered across **Math**, **Code**, **Translation**, and **General** Conversation.

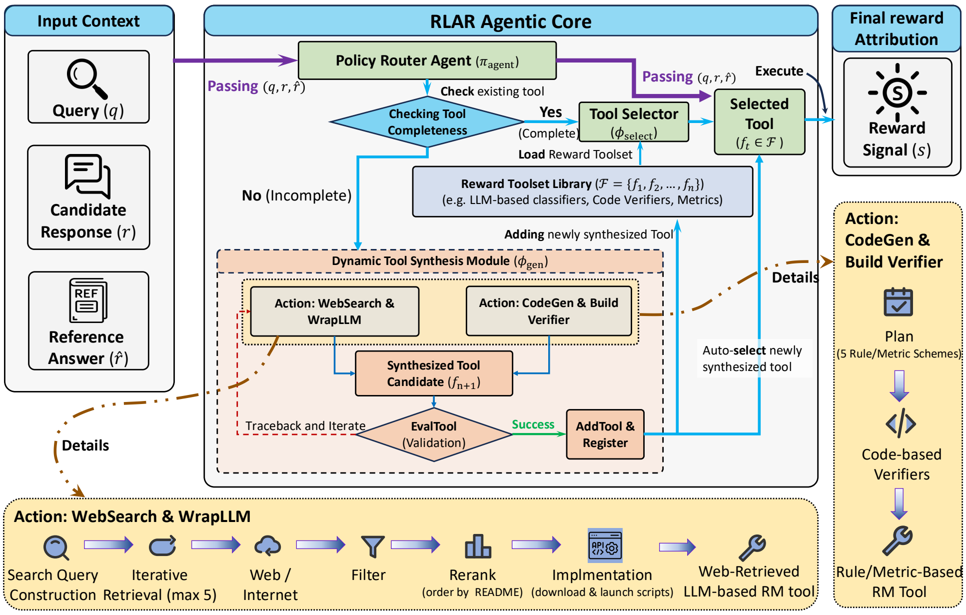


Figure 2: The Reinforce Learning with Agentic Reward (RLAR) Framework.

3 Methodology

To address the inherent limitations of monolithic reward models in handling heterogeneous task distributions, we propose **Reinforcement Learning with Agentic Reward** (RLAR). Unlike static reward functions, RLAR is a self-evolving framework that leverages the tool-use capabilities of LLMs to dynamically synthesize, verify, and invoke specialized reward tools.

3.1 Problem Formulation

We formalize the RLAR framework as a dynamic reward synthesis and calling problem. Given an input context triplet $(q, r, \hat{r})$ comprising a query, the candidate response, and the reference answer, our objective is to compute a precise scalar reward $s \in \mathbb{R}$ that reflects the response quality.

The framework operates on a **Reward Toolset (Library)** $\mathcal{F} = \{f_1, f_2, \dots, f_n\}$, where each $f_i : (q, r, \hat{r}) \rightarrow s$ is an atomic reward tool (e.g., an LLM-classifier-based Reward Model, a code-based checker, or a heuristic rule). Initially, $\mathcal{F}$ is populated with a general-purpose base reward model (e.g., Skywork-Llama). As the framework encounters more queries from diverse tasks, $\mathcal{F}$ expands through agentic synthesis: $\mathcal{F}_{t+1} = \mathcal{F}_t \cup \{f_{new}\}$.

3.2 Adaptive Reward Routing and Selection

A core challenge in RLAR is determining whether the current toolset $\mathcal{F}$ can accurately evaluate a given $(q, r, \hat{r})$. We introduce a **Policy Router** π_{agent}, which serves as the decision-making core.

Completeness Assessment: For each incoming $(q, r, \hat{r})$, π_{agent} assesses the *competency* of the existing library. We formalize this as a classification-of-intent task:

$$a = \pi_{agent}(\text{state} = (q, r, \hat{r}), \mathcal{F}) \qquad (1)$$

where $a \in \{\text{Select}, \text{Synthesize}\}$.

Tool Selection: If $\mathcal{F}$ is complete ($a = \text{Select}$), a selection module ϕ_{select} identifies the optimal reward tool $f_t \in \mathcal{F}$ by matching the task metadata with tool descriptions.

Tool Synthesis: If no suitable tool exists ($a =$ Synthesize), it calls the Agentic Synthesis module to construct the missing reward tool.

3.3 Agentic Tool Synthesis

When a tool is missing from the reward tool library $\mathcal{F}$, RLAR invokes specialized agents to construct a new reward tool f_{new}. Considering different reward system types, we designed **WrapLLM** and **CodeVerify** to fulfill this goal.

3.3.1 WrapLLM

The LLM agent retrieves and encapsulates the most compatible lightweight (within 10B) Reward Model repository from the open-sourced model platform *HuggingFace*. They suit tasks requiring semantic nuances, creativity, or value alignment. The agent follows an iterative retrieval workflow: (1) Generate a search query for the search engine (2) Iteratively retrieving from the search engine (3) Filter and rerank results for the most related reward model (4) Download and implement the deployment and inference code (5) Wrap (generating the function names, descriptions) into an API for tool calling. The workflow transforms a reward model checkpoint into a callable tool for $\mathcal{F}$.

3.3.2 CodeVerify

The agent follows a Plan-and-Execute strategy to generate deterministic verifiers for objective tasks such as mathematics and programming. It would initially plan at most 5 possible schemes and implement the optimal one into reward tool.

The agent generates Python-based scripts that vary from task type. For example, retrieving the final answer from a formatted solution as a reward tool is suited for math; retrieving and arranging generated code and executing it with test cases would be suited for code. Furthermore, for tasks such as translation, it would implement a metric (e.g., ROUGE, BLEU, METEOR) based reward tool. The prompts for the above are provided in Appendix D.4.

3.4 Verification and Library Update

To ensure the reliability of the synthesized tools, each f_{new} must pass a module (**EvalTool**) before being committed to $\mathcal{F}$. For *WrapLLM*, it examines the consistency between the model-as-a-tool description and its original documentation. For *CodeVerify*, it performs analysis on code implementation and execution checks on synthesized scripts. Only tools that satisfy these verifications are committed to the library: $\mathcal{F} \leftarrow \mathcal{F} \cup \{f_{new}\}$.

4 Experiments

4.1 Dataset and Evaluation

Math Reasoning: We use GSM8K (Cobbe et al., 2021) and HENDRYCKS-MATH (Hendrycks et al., 2021) for mathematical reasoning. The goal is to reason for the final numeric solution.

Coding: We select the LEETCODE-DATASET (Xia et al., 2025), which asks the testers to fill in the LeetCode cloze-style function implementation.

Translation: We sample from FLORES-101 (Goyal et al., 2021) to cover 20 translation pairs across five languages (CN, DE, EN, FR, JP, in alphabetical order). This task requires mastery across languages and is important for LLMs' accessibility to worldwide users, especially non-English speakers.

General Alignment: ULTRACHAT (Ding et al., 2023) is employed to train and evaluate multi-turn dialogue and instruction-following capabilities.

Generalization Challenge. To assess the robustness of the trained policy, we applied benchmarks including AIME-24/25 for math reasoning, MBPP (Austin et al., 2021) for Python synthesis, and WMT-24 for unseen translation contexts. These benchmarks evaluate that the post-trained model can generalize rather than over-fit to the specific heuristics of the training reward models, as an indication of reward hacking.

Data Process. For all queries in the training dataset split, we applied an automated quality filtering process, requiring the LLM to remove samples that did not meet the standards based on both query quality and response quality. The prompt is provided in Appendix B.2. Then we uniformly downsample over GSM8K, HENDRYCKS-MATH, FLORES-200, WMT-24, ULTRA-CHAT to ensure a balanced distribution of queries across different dataset sources. To facilitate our experiments, we obtained the reasoning outputs of GPT-5 on all queries. For dataset queries lacking human-annotated reasoning references, we supplemented them with the results from GPT-5. Table 1 shows the statistics of the train and validation sets. A more detailed introduction of the task and our selected dataset is listed in Appendix B.

Evaluation Metrics. To ensure evaluation rigor, we apply Numeric Exact Match (**NEM**) at Best@1 for mathematical reasoning tasks. For code generation, we report **pass@1** verified by automated code executors. For translation tasks, we report both **BLEU** (Papineni et al., 2002) and **LLM-judge** scores, while general dialogue tasks are evaluated solely by the LLM-judge. Additionally, we measure the **average generation lengths** across all benchmarks to detect potential reward-hacking behaviors associated with the verbosity bias inherent in LLM-based rewarding. GPT-5 is utilized as the primary evaluator for all LLM-as-a-judge assessments, and the 0-10 scoring range is normalized to 0-100 in accordance with other metrics. We provide an **average score** computed by averaging individual benchmark results. Specifically, for the translation benchmark, the score is a composite of 0.5 BLEU-2 and 0.5 LLM-judge.

4.2 Experimental Setup

Base Models: We use Qwen3-8B (Yang et al., 2025) and Llama-3.1-8B-Instruct (Weerawardhena et al., 2025) as base models. To examine various reward system architectures, we incorporate the following baselines:

Classification Reward Baselines: For the single classification reward model setting, we select `Skywork-Reward-V2-Llama-3.1-8B` (**SkyLlama**) and `Skywork-Reward-V2-Qwen3-8B` (**SkyQwen**) (Liu et al., 2024), both of which represent the state-of-the-art on REWARDBENCH-V1/V2 (Lambert et al., 2024; Malik et al., 2025a). Additionally, we incorporate `Seed-X-8B-RM` (**SeedX**) due to its robust performance in multilingual contexts (Cheng et al., 2025).

Generative Reward Baselines: We evaluate the `Skywork/Skywork-Critic-Llama-3.1-8B` (**SkyCritic**) (Tu et al., 2024) , which generates natural language justifications with its judgments. Furthermore, we employ GPT-5-as-a-judge (**GPT5-judge**) as a high-capacity competitor, following the RL from AI Feedback paradigm (Lee et al., 2024). In this setup, GPT-5 is prompted to act as an evaluator, and its numerical scores are extracted as reward signals. The specific prompt template is detailed in Appendix C.2.

Human Baseline: We implement a task-specific reward system to calibrate performance against optimal human-defined heuristics. For each task, the reward function is defined by the metrics most appropriate for the domain. Specifically, we utilize **NEM** for mathematics:

$$R_{\text{math}}(r, \hat{a}) = \begin{cases} 1, & \text{if NEM}(r, \hat{a}) \\ 0, & \text{otherwise} \end{cases}$$

where $\hat{a}$ denotes the ground-truth numerical answer, and **pass@1** for coding:

$$R_{\text{code}}(r) = \begin{cases} 1, & \text{if Exec}(r) = \text{Pass} \\ 0, & \text{otherwise} \end{cases}$$

For translation, the reward is calculated as a hybrid score $w_1 \cdot \text{BLEU} + w_2 \cdot \text{SeedX}$, where the scaling factor is $w1 = 1/2, w_2 = 1/2$ and SeedX's score is normalized to $(0, 1)$. The dialogue tasks are rewarded only by **SkyLlama**. We have attached the **training configurations** in Appendix C.1 and **hyperparameter** for training in Appendix C.3. RLAR is powered by GPT-5 by default.

4.3 Main Results

Superiority of RLAR across Mixed Domains. RLAR consistently outperforms both classification-based and generative reward baselines across multiple tasks and backbone models (average score $+10.4\%$ on Llama and $+61.9\%$ on Qwen). Notably, RLAR achieves the best results in the majority of benchmarks (73.6 NEM on GSM8k for Llama-3.1 and 23.3 Pass@1 on LeetCode for Qwen3) without incurring significant performance degradation in other data distributions. The increase can also be found in the OOD benchmarks AIME-24(+16.6), AIME-25(+13.3), AIME-24(+5.03) and WMT-24(+3.41 BLEU). In contrast, **SkyLlama**, **SkyQwen**, **SeedX** suffer from catastrophic failures when faced with Qwen3 on GSM-8K (-31.9) and ULTRACHAT while they exhibits benefits to HENDYRCKS and LEETCODE. This indicates that RLAR maintains stable and generalizable performance by dynamically modeling domain objectives into new reward function designs without neglecting specific distributions.

The results from Llama-3.1 indicate that RL post-training might not be beneficial, probably due to its near-saturated base model performance (60.5 on GSM8k, 41.17 on MBPP). However, RLAR still causes most increases on most in-distribution and OOD test-benches (+13.1 on GSM-8k, +3.41 on WMT evaluated by BLEU). **SkwLlama**, **SkyQwen**, **SkyCritic** mostly failed on GSM8k, causing catastrophic performance loss ($-49.2, -44.3, -15.9$). They also rarely show benefits on OOD benchmarks (AIME, MBPP, WMT),

Domain	Avg.	Math				Code		Translation				General	
Dataset	**Score**	GSM8k	HEND	**AIME24**	**AIME25**	LCode	**MBPP**	Flore		**WMT24**		UChat	Avg Len
Metric	-	NEM	Pass@k	NEM	NEM	Pass@k	Pass@k	BLEU-2	LLM-J	BLEU-2	LLM-J	LLM-J	#
							Llama-3.1-8B						
Base	37.81	60.50	49.80	6.67	0.00	11.21	41.17	21.04	77.27	26.39	79.24	68.97	152.84
SkyLlama	23.47	11.30 ↓49.20	47.60 ↓2.20	3.33 ↓3.34	0.00 -	8.97 ↓2.24	7.80 ↓33.37	21.16 ↑0.12	45.32 ↓31.95	23.23 ↓3.16	54.68 ↓24.56	60.03 ↓8.94	147.39
SeedX	17.29	1.21 ↓59.29	42.20 ↓7.60	6.67 -	0.00 -	0.00 ↓11.21	13.76 ↓27.41	0.00 ↓21.04	23.24 ↓54.03	0.00 ↓26.39	42.76 ↓36.48	58.73 ↓10.24	89.95
SkyQwen	31.68	16.15 ↓44.35	46.60 ↓3.20	3.33 ↓3.34	0.00 -	**11.66 ↑0.45**	36.04 ↓5.13	26.42 ↑5.38	77.53 ↑0.26	29.80 ↑3.41	78.74 ↓0.50	65.13 ↓3.84	134.21
SkyCritic	34.05	44.58 ↓15.92	48.00 ↓1.80	10.00 ↑3.33	0.00 -	10.31 ↓0.90	**45.69 ↑4.52**	25.31 ↑4.27	42.53 ↓34.74	28.21 ↑1.82	67.82 ↓11.42	65.93 ↓3.04	106.97
GPT5-Judge	39.89	71.95 ↑11.45	43.20 ↓6.60	6.67 -	6.67 ↑6.67	7.17 ↓4.04	45.17 ↑4.00	20.35 ↓0.69	**82.50 ↑5.23**	26.37 ↓0.02	**84.42 ↑5.18**	**71.32 ↑2.35**	203.31
Human	33.56	42.38 ↓18.12	40.00 ↓9.80	3.33 ↓3.34	3.33 ↑3.33	10.31 ↓0.90	37.37 ↓3.80	26.61 ↑5.57	70.67 ↓6.60	**29.81 ↑3.42**	75.88 ↓3.36	63.87 ↓5.10	76.76
RLAR(Ours)	**41.73**	**73.61 ↑13.11**	**54.60 ↑4.80**	**13.33 ↑6.66**	**6.67 ↑6.67**	**11.66 ↑0.45**	43.18 ↑2.01	**26.73 ↑5.69**	74.83 ↓2.44	29.80 ↑3.41	79.54 ↑0.30	67.03 ↓1.94	141.85
							Qwen3-8B						
Base	29.12	31.99	19.80	3.33	0.00	6.28	14.27	28.46	84.00	32.50	84.46	71.70	1064.04
SkyLlama	29.25	0.00 ↓31.99	35.80 ↑16.00	10.00 ↑6.67	6.67 ↑6.67	7.62 ↑1.34	14.85 ↑0.58	28.78 ↑0.32	85.59 ↑1.59	33.75 ↑1.25	86.54 ↑2.08	71.00 ↓0.70	844.96
SeedX	35.14	0.00 ↓31.99	**72.60 ↑52.80**	6.67 ↑3.34	6.67 ↑6.67	21.08 ↑14.80	20.53 ↑6.26	30.34 ↑1.88	84.97 ↑0.97	35.25 ↑2.75	85.14 ↑0.68	70.83 ↓0.87	1135.82
SkyQwen	33.48	0.00 ↓31.99	67.00 ↑47.20	3.33 -	3.33 ↑3.33	15.70 ↑9.42	21.25 ↑6.98	29.70 ↑1.24	85.72 ↑1.72	33.82 ↑1.32	85.18 ↑0.72	73.47 ↑1.77	1192.12
SkyCritic	39.07	51.71 ↑19.72	58.00 ↑38.20	**20.00 ↑16.67**	6.67 ↑6.67	13.00 ↑6.72	12.32 ↓1.95	28.66 ↑0.20	86.09 ↑2.09	32.52 ↑0.02	86.46 ↑2.00	73.10 ↑1.40	691.97
GPT5-Judge	45.85	91.96 ↑59.97	56.20 ↑36.40	16.67 ↑13.34	10.00 ↑10.00	21.52 ↑15.24	18.17 ↑3.90	28.31 ↓0.15	**89.77 ↑5.77**	32.64 ↑0.14	**88.93 ↑4.47**	**78.32 ↑6.62**	1462.29
Human	44.40	**95.30 ↑63.31**	53.80 ↑34.00	**20.00 ↑16.67**	10.00 ↑10.00	8.52 ↑2.24	18.89 ↑4.62	30.71 ↑2.25	85.11 ↑1.11	35.91 ↑3.41	85.22 ↑0.76	74.63 ↑2.93	692.71
RLAR(Ours)	**47.16**	93.70 ↑61.71	59.80 ↑40.00	**20.00 ↑16.67**	**13.33 ↑13.33**	**23.31 ↑17.03**	19.30 ↑5.03	**31.12 ↑2.66**	86.49 ↑2.49	**36.07 ↑3.57**	86.67 ↑2.21	74.81 ↑3.11	986.23

Table 2: **Main Experiment Results**. For each testset, **Bold** indicates the best result under that metric, and underline indicates the second best within each model group. ↑Increment and ↓decrement are calculated relative to the Zero-Shot baseline for Llama and Qwen. **LLM-J** is short for LLM-judge, and **NEM** shorts for numeric exact match. **HEND** shorts for HENDRYCKS-MATH, **LCode** shorts for LEETCODE-DATASET, **UChat** shorts for ULTRA-CHAT. The bold benchmarks (**AIME24/25, MBPP, WMT24**) indicate that they do not include the training data.

indicating the performance gain is less often generalized outside the training domains. **GPT5-Judge** optimized LLama on GSM8K by 11.4, while causing degrades on HENDRYCKS-MATH by 6.60 and LEETCODE by 4.04. These demonstrate RLAR's robustness to avoid performance degradation, benefiting from its more comprehensive reward design.

Comparison with GPT5-Judge and Human Baseline. While **GPT5-judge** serves as a competitive baseline, it exhibits vulnerabilities in rule-heavy tasks. For example, on Llama-3.1, GPT5-Judge triggers performance drops on Hendrycks-Math and LeetCode compared to zero-shot levels (-6.60/-4.04) while RLAR does not ($+4.80$/$+0.45$), suggesting that reward systems lacking explicit rule verification are susceptible to "blind spots" in deterministic domains. Furthermore, RLAR demonstrates significant advantages in **cost-effectiveness and training efficiency**: a full 80-step RL training session with GPT-5 costs approximately 120 million API tokens, and the training takes over GPU 288 hours, whereas RLAR reduces them to 23.7 million API tokens and 72 GPU hours. RLAR represents a highly efficient and economical paradigm for RLAIF.

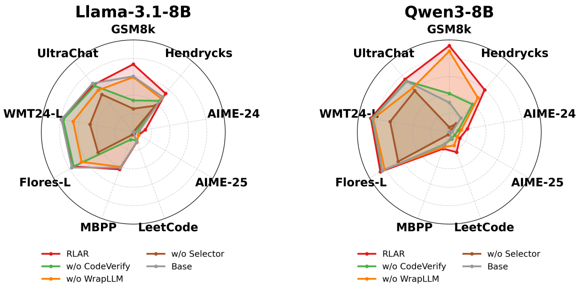
Intriguingly, RLAR even surpasses the Human baseline in several reasoning tasks (Math and Code benchmarks on Llama, HENDRYCKS-MATH, LEETCODE on Qwen). Qualitative analysis reveals that RLAR's reward signals provide more comprehensive coverage than simple heuristics, incorporating additional test cases and rigorous mathematical formatting checks. This effectively increases the completeness of the rule-based verifiers.

4.4 Reward Hacking Identification

Hack with Format. A critical failure mode observed in classification-based RM baselines on Qwen3-8B is "Format Hacking." Specifically, all three classification baselines (**SkyLlama, SeedX, SkyQwen**) collapsed to a 0.00 score on GSM8K. Manual inspection reveals that the policy models learned to bypass task-specific constraints: while the instructions required "####" before GSM8K answers and "`\boxed{}`" for HENDRYCKS-MATH, the models mistakenly unified all outputs into the Hendrycks format. Thus, baselines may overlook subtle instructional requirements, allowing the policy to 'exploit' the reward system by mimicking specific output formats rather than optimizing the actual reward. Our method exhibits robustness on such loopholes, as the trained policy can easily handle the formats across all benchmarks.

Hack with Verbosity. Furthermore, we observe a persistent verbosity bias in LLM-based rewards baselines. For example, models trained with GPT-5-Judge consistently produced outputs 30% **longer** than those partly learned from rule-based rewards (Human/RLAR). This confirms that verbosity bias is inherently latent in reward LLMs and is potentially triggered during RL. Due to the rewarding versatility and the abundant included verifiable rewards, RLAR did not introduce a significant long-tail generation problem.

4.5 Ablations on Modules



(a) Llama-3.1-8B (b) Qwen3-8B

Figure 3: Ablation studies on Llama-3.1 and Qwen3.

To investigate the contributions of the actions within the RLAR framework, we conduct an ablation study with the following configurations: (1) **w/o CodeVerify** and (2) **w/o WrapLLM**, which respectively disable the corresponding actions in the tool synthesis module; and (3) **w/o Selector**, where instead of immediate selection post-synthesis, a reward tool is randomly sampled only after all reward tools have been generated. The comparative results of these variants are visualized in Figure 3.

Removing the components of the tool synthesis module consistently results in degradation. Specifically, **without CodeVerify** designed tools, mathematical reasoning and coding task (GSM8k, LeetCode) drop significantly. **WrapLLM** is essential for chat and translation tasks. The above results confirm the necessity of query-specific reward designs, which is the core motivation of RLAR.

Moreover, the Random Reward Selection policy variant (**w/o Selector**) emphasizes that reward matching is important. The policy degrades on almost all testbenches. This confirms that the appropriate reward system design is crucial in the holistic procedure of RL training, and that our framework effectively synthesizes high-quality reward tools superior to random alternatives.

4.6 Evaluating Reward Routing and Selection

The core mechanism of the RLAR framework involves predicting and selecting the optimal reward function for a specific query triplet. As shown in the previous section, such a selection strategy is crucial for the performance. To demonstrate that RLAR achieves superior performance through the designed adaptive reward routing and selection, we conducted an analytical experiment using RewardBench-v2.

Setup. We utilized a randomly and uniformly sampled subset of 400 instances from the RewardBench-v2 test set. Each instance comprises one preferred (chosen) response and three non-preferred (rejected) responses for a given prompt.[2]. For each instance, a reward model is required to score all four responses. An instance is considered a **pass** if the softmax-normalized reward score assigned to the preferred response exceeds a threshold of 0.5.

We leveraged the evaluation records of the top 50 classification models from the official leaderboard. RLAR is tasked with predicting the reward model most likely to pass a given instance, utilizing the query, candidate responses, and the models' metadata. For evaluation simplicity, reward models are presented to the agent as encapsulated tools. We compared the following approaches: **Backbone LLM Scaling**: Evaluating the impact of different agentic capabilities using GPT-4o, GLM-4.6, and GPT-5.1. **Reward Logits Merging**: Assessing the effect of ensemble methods by calculating the average scores of the top-k models' predictions on each candidate responses (mean@k).

Results and Discussion. As shown in Table 3, simply merging reward model output logits (mean@k) does not effectively expand the performance ceiling; in fact, performance degrades significantly as more models are included (mean@50). In contrast, there is a clear positive correlation between the **agentic capability** of the backbone LLM and performance gains. These findings demonstrate that leveraging LLM reasoning to dynamically select reward models is a viable path for extending reward system design, approaching the theoretical performance limit of the available model pool.

We have further conducted the analysis on the WrapLLM search engine design. The average retrieved model repository position is 5.64 in the first page, showing the relatedness. **Further error analyses are included in Appendix** G.1.

5 Related Work

5.1 LLM RL Reward Designs

In industry, training **discriminative** reward models (Ouyang et al., 2022; DeepSeek-AI et al., 2025; Liu et al., 2024) is widely regarded as the most

---

[2] The "Tie" category is excluded due to the test's input-output format uniformity.

Method	Avg	Precision IF	Math	Safety	Factuality	Focus
SOTA	87.19	57.14	60.00	97.12	85.29	99.13
Agentic Selection within RLAR *framework*						
Random	78.39	40.48	60.00	89.42	75.49	90.43
GPT-4o	88.19	57.14	68.57	96.15	87.25	99.13
GLM-4.6	88.69	57.14	68.57	96.15	88.24	100.0
GPT-5.1	**90.44**	**66.67**	**71.42**	96.15	**90.19**	**100.0**
Logits Merging Methods						
mean@2	87.44	61.90	68.57	95.19	83.33	99.13
mean@10	77.89	47.62	57.14	84.62	76.47	90.43
mean@50	51.51	40.48	48.57	66.35	48.04	46.09
Theoretical Best	93.47	76.19	77.14	97.12	95.10	100.0

Table 3: Comparison of Agentic Selection vs. Reward Merging on REWARDBENCH-V2.

reliable approach for constructing a human preference oracle within reinforcement learning (RL) frameworks for LLM optimization. In addition, **generative** rewards extend the aforementioned task from classification to generation, and have demonstrated feasibility in mathematical domains (Zhang et al., 2025a), RLHF-based settings (Ke et al., 2024; Wang et al., 2024b; Zhu et al., 2025; Li et al., 2023), and can be integrated with advances in LLM reasoning, such as CritiqueGRPO (Zhang et al., 2025b). With the rapid development of math reasoning and code generation, the design of **verifiable** rewards has attracted increasing attention. Binary rewards that can be verified through explicit rules have been shown to be more efficient in these domains (Shao et al., 2024; Lambert et al., 2025). Chang et al. (2025) involves employing standard NLP metrics to guide instruction-following alignment as an extension of aforementioned approach.

5.2 RL from AI Feedback

RLAIF (Lee et al., 2024) explores the development of reward models without extensive manual labeling of training data. Self-rewarding (Yuan et al., 2025) require the policy model to evaluate and discriminate its own generations. The LLM-as-a-judge (Zheng et al., 2023) paradigm employs a strong LLM to evaluate another LLM by means of a preceding evaluation prompt. RewardAgent (Peng et al., 2025) utilizes an LLM to combine pre-specified reward designs. These approaches inevitably embed strong human priors into reward design, either through the evaluation prompt or through the foundational reward specifications. In contrast to RewardAgent, our work extends both the design flexibility and the evaluation of reward design framework within an RL framework where reward is explicitly formulated (specifically GRPO rather than DPO).

5.3 Dynamic Reward Assigning

Recent research in integrating LLM with RL, particularly for reward shaping, has primarily focused on analyzing the agent's policy trace from prior steps to iteratively refine the reward function. (Afonso et al., 2025) and (Carta et al., 2022) leverage the LLM's reasoning to guide reward weight pruning or analyze the trace to determine the appropriate reward shape design. Other methods, such as (Xie et al., 2025) and (Singla et al., 2024) explore techniques like curriculum scheduling and adjusting the reward schedule via prompt hints. RLAR diverges significantly by harnessing the LLM's capability to search the web and generate code, allowing it to directly **design entirely new rewards** rather than being limited to weight adjustments. RLAR is also flexible for **cross-domain optimization** problems, where reward designs differ substantially across various sub-domains, a challenge that existing single-task-focused methods do not fully address.

6 Conclusion

In this paper, we present RLAR, a novel agentic rewarding framework that designs query-specific reward functions for RL algorithm. By integrating a self-evolving tool library with dynamic synthesis and selection, RLAR achieves superior performance across diverse domains, including math, code, translation and general chat. Beyond empirical gains, RLAR demonstrates strong robustness against format and verbosity reward hacking. By significantly reducing the costs compared to RLAIF practices, RLAR establishes the "LLM as Reward Designer" paradigm.

7 Limitations

We primarily validated RLAR on heterogeneous tasks in text forms. Due to the budget constraints, we did not extend the scope into multi-modal, audio tasks such as text-to-image generations. We believe this is a good exploration field for future works. There is still room for further analysis on the scalability of the RLAR framework.

In practice, some repository README would become out-dated when reporting (such as claiming to be state-of-the-art of that time). Though not directly caused by the design, RLAR is potentially vulnerable to readme hacking, as our assumption is that most of these repo readmes are trustworthy. We leave the development for developing more robust retrieval modules for future works.

Lastly, we focus on language models that are modeled as text classifiers. This is quite similar to practices in the industry, mainly aiming to save the computational cost of reward calculation. For generative reward models, our framework can support development on this basis; however, given the constraints of our experimental setup, we consider this to be outside the scope of the present work.

References

António Afonso, Iolanda Leite, Alessandro Sestini, Florian Fuchs, Konrad Tollmar, and Linus Gisslén. 2025. Self-correcting reward shaping via language models for reinforcement learning agents in games. Preprint, arXiv:2506.23626.

Mayank Agarwal, Ibrahim Abdelaziz, Kinjal Basu, Merve Unuvar, Luis A. Lastras, Yara Rizk, and Pavan Kapanipathi. 2025. Toolrm: Outcome reward models for tool-calling large language models. Preprint, arXiv:2509.11963.

Agentlans. 2019. allenai-wildchat-1m-multiturn. `agentlans/allenai-WildChat-1M-multiturn`.

Aircrypto. 2019. English-french-translations-train-large. `aircrypto/English-French-Translations-Train-Large`.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. Preprint, arXiv:2108.07732.

Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, Xiaoyi Dong, Haodong Duan, Qi Fan, Zhaoye Fei, Yang Gao, Jiaye Ge, Chenya Gu, Yuzhe Gu, Tao Gui, and 81 others. 2024. Internlm2 technical report. Preprint, arXiv:2403.17297.

Thomas Carta, Pierre-Yves Oudeyer, Olivier Sigaud, and Sylvain Lamprier. 2022. Eager: Asking and answering questions for automatic reward shaping in language-guided rl. Preprint, arXiv:2206.09674.

Yapei Chang, Yekyung Kim, Michael Krumdick, Amir Zadeh, Chuan Li, Chris Tanner, and Mohit Iyyer. 2025. Bleuberi: Bleu is a surprisingly effective reward for instruction following. Preprint, arXiv:2505.11080.

Shanbo Cheng, Yu Bao, Qian Cao, Luyang Huang, Liyan Kang, Zhicheng Liu, Yu Lu, Wenhao Zhu, Jingwen Chen, Zhichao Huang, Tao Li, Yifu Li, Huiying Lin, Sitong Liu, Ningxin Peng, Shuaijie She, Lu Xu, Nuo Xu, Sen Yang, and 7 others. 2025. Seed-x: Building strong multilingual translation llm with 7b parameters. Preprint, arXiv:2507.13618.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. Preprint, arXiv:2110.14168.

Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. 2018. A discourse-aware attention model for abstractive summarization of long documents. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers), pages 615–621, New Orleans, Louisiana. Association for Computational Linguistics.

Nicholas Kluge Corrêa. 2023. Aira.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. Preprint, arXiv:2501.12948.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 3029–3051, Singapore. Association for Computational Linguistics.

Naman Goyal, Cynthia Gao, Vishrav Chaudhary, Peng-Jen Chen, Guillaume Wenzek, Da Ju, Sanjana Krishnan, Marc'Aurelio Ranzato, Francisco Guzmán, and Angela Fan. 2021. The flores-101 evaluation benchmark for low-resource and multilingual machine translation.

Francisco Guzmán, Peng-Jen Chen, Myle Ott, Juan Pino, Guillaume Lample, Philipp Koehn, Vishrav

Chaudhary, and Marc'Aurelio Ranzato. 2019. Two new evaluation datasets for low-resource machine translation: Nepali-english and sinhala-english.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. Preprint, arXiv:2103.03874.

Xu Huang, Wenhao Zhu, Hanxu Hu, Conghui He, Lei Li, Shujian Huang, and Fei Yuan. 2025. Benchmax: A comprehensive multilingual evaluation suite for large language models. Preprint, arXiv:2502.07346.

Pei Ke, Bosi Wen, Andrew Feng, Xiao Liu, Xuanyu Lei, Jiale Cheng, Shengyuan Wang, Aohan Zeng, Yuxiao Dong, Hongning Wang, Jie Tang, and Minlie Huang. 2024. CritiqueLLM: Towards an informative critique generation model for evaluation of large language model generation. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 13034–13054, Bangkok, Thailand. Association for Computational Linguistics.

Michael Krumdick, Charles Lovering, Varshini Reddy, Seth Ebner, and Chris Tanner. 2025. No free labels: Limitations of llm-as-a-judge without human grounding. Preprint, arXiv:2503.05061.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, and 4 others. 2025. Tulu 3: Pushing frontiers in open language model post-training. Preprint, arXiv:2411.15124.

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. 2024. Rewardbench: Evaluating reward models for language modeling. Preprint, arXiv:2403.13787.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. 2024. RLAIF: Scaling reinforcement learning from human feedback with AI feedback.

Dawei Li, Renliang Sun, Yue Huang, Ming Zhong, Bohan Jiang, Jiawei Han, Xiangliang Zhang, Wei Wang, and Huan Liu. 2025a. Preference leakage: A contamination problem in llm-as-a-judge. Preprint, arXiv:2502.01534.

Junlong Li, Shichao Sun, Weizhe Yuan, Run-Ze Fan, Hai Zhao, and Pengfei Liu. 2023. Generative judge for evaluating alignment. Preprint, arXiv:2310.05470.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. 2025b. CodePRM: Execution feedback-enhanced process reward model for code generation. In Findings of the Association for Computational Linguistics: ACL 2025, pages 8169–8182, Vienna, Austria. Association for Computational Linguistics.

Chris Yuhao Liu, Liang Zeng, Jiacai Liu, Rui Yan, Jujie He, Chaojie Wang, Shuicheng Yan, Yang Liu, and Yahui Zhou. 2024. Skywork-reward: Bag of tricks for reward modeling in llms. Preprint, arXiv:2410.18451.

Saumya Malik, Valentina Pyatkin, Sander Land, Jacob Morrison, Noah A. Smith, Hannaneh Hajishirzi, and Nathan Lambert. 2025a. Rewardbench 2: Advancing reward model evaluation. Preprint, arXiv:2506.01937.

Saumya Malik, Valentina Pyatkin, Sander Land, Jacob Morrison, Noah A. Smith, Hannaneh Hajishirzi, and Nathan Lambert. 2025b. Rewardbench 2: Advancing reward model evaluation. Preprint, arXiv:2506.01937.

James Cross Onur Çelebi Maha Elbayad Kenneth Heafield Kevin Heffernan Elahe Kalbassi Janice Lam Daniel Licht Jean Maillard Anna Sun Skyler Wang Guillaume Wenzek Al Youngblood Bapi Akula Loic Barrault Gabriel Mejia Gonzalez Prangthip Hansanti John Hoffman Semarley Jarrett Kaushik Ram Sadagopan Dirk Rowe Shannon Spruit Chau Tran Pierre Andrews Necip Fazil Ayan Shruti Bhosale Sergey Edunov Angela Fan Cynthia Gao Vedanuj Goswami Francisco Guzmán Philipp Koehn Alexandre Mourachko Christophe Ropers Safiyyah Saleem Holger Schwenk Jeff Wang NLLB Team, Marta R. Costa-jussà. 2022. No language left behind: Scaling human-centered machine translation.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. 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 Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. Preprint, arXiv:2203.02155.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Hao Peng, Yunjia Qi, Xiaozhi Wang, Zijun Yao, Bin Xu, Lei Hou, and Juanzi Li. 2025. Agentic reward modeling: Integrating human preferences with verifiable correctness signals for reliable reward systems. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 15934–15949, Vienna, Austria. Association for Computational Linguistics.

Qwedsacf. 2019. ivypanda-essays. qwedsacf/ivypanda-essays.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. Preprint, arXiv: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. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. Preprint, arXiv:2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. arXiv preprint arXiv: 2409.19256.

Somanshu Singla, Zhen Wang, Tianyang Liu, Abdullah Ashfaq, Zhiting Hu, and Eric P. Xing. 2024. Dynamic rewarding with prompt optimization enables tuning-free self-alignment of language models. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 21889–21909, Miami, Florida, USA. Association for Computational Linguistics.

Shiwen Tu, Liang Zhao, Liu Chris Yuhao, Zeng Liang, and Liu Yang. 2024. Skywork critic model series. https://huggingface.co/Skywork.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and Zhifang Sui. 2024a. Large language models are not fair evaluators. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 9440–9450, Bangkok, Thailand. Association for Computational Linguistics.

Yidong Wang, Zhuohao Yu, Wenjin Yao, Zhengran Zeng, Linyi Yang, Cunxiang Wang, Hao Chen, Chaoya Jiang, Rui Xie, Jindong Wang, Xing Xie, Wei Ye, Shikun Zhang, and Yue Zhang. 2024b. PandaLM: An automatic evaluation benchmark for LLM instruction tuning optimization. In The Twelfth International Conference on Learning Representations.

Sajana Weerawardhena, Paul Kassianik, Blaine Nelson, Baturay Saglam, Anu Vellore, Aman Priyanshu, Supriti Vijay, Massimo Aufiero, Arthur Goldblatt, Fraser Burch, Ed Li, Jianliang He, Dhruv Kedia, Kojin Oshiba, Zhouran Yang, Yaron Singer, and Amin Karbasi. 2025. Llama-3.1-foundationai-securityllm-8b-instruct technical report. Preprint, arXiv:2508.01059.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. 2025. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. Preprint, arXiv:2504.14655.

Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. 2025. Logic-rl: Unleashing llm reasoning with rule-based reinforcement learning. Preprint, arXiv:2502.14768.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. Preprint, arXiv:2505.09388.

Rui Yang, Xiaoman Pan, Feng Luo, Shuang Qiu, Han Zhong, Dong Yu, and Jianshu Chen. 2024. Rewards-in-context: Multi-objective alignment of foundation models with dynamic preference adjustment. International Conference on Machine Learning.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2025. Self-rewarding language models. Preprint, arXiv:2401.10020.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. 2025a. Generative verifiers: Reward modeling as next-token prediction. Preprint, arXiv:2408.15240.

Xiaoying Zhang, Hao Sun, Yipeng Zhang, Kaituo Feng, Chaochao Lu, Chao Yang, and Helen Meng. 2025b. Critique-grpo: Advancing llm reasoning with natural language and numerical feedback. Preprint, arXiv:2506.03106.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2025c. The lessons of developing process reward models in mathematical reasoning. In Findings of the Association for Computational Linguistics: ACL 2025, pages 10495–10516, Vienna, Austria. Association for Computational Linguistics.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-bench and chatbot arena. In Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track.

Lianghui Zhu, Xinggang Wang, and Xinlong Wang. 2025. JudgeLM: Fine-tuned large language models are scalable judges. In The Thirteenth International Conference on Learning Representations.

A Reproducibility Statement

We have included the heterogeneous data construction process in Section 2.2 and more details in Appendix. We described the RL training settings and experiment platforms in Appendix C.3 and Section ??. The prompts involving the usage of LLM (primarily GPT-4.1) are filed in Appendix D. The above materials are able to reproduce our work.

B Data Process Details

B.1 Detailed Introduction of Datasets

Translation (En-Fr, Fr-En): This task requires the LLM to translate between English and French (in our case, English to French and French to English). We use the dataset `aircrypto/English-French-Translations-Train-Large` (Aircrypto, 2019) from HuggingFace, which provides high-quality, paired sentence-level samples.

Instruction Following: Given specific requirements in the provided instructions, the LLM should respond accordingly. We use `tulu3-sft-reused-on-policy-8b`, part of the Tulu-3 (Lambert et al., 2025) preference dataset, which contains generation pairs between different LLMs during the training of Llama-3.1-Tulu-3-8B.

Multi-turn: LLM respond to instructions with previous interaction histories. We pick `allenai-WildChat-1M-multiturn` (Agentlans, 2019), a collection of 1M ChatGPT interaction logs from the wild. We select the English subset aimed at RLHF queries.

Summarization: This task requires LLM to summarize over long documents into short abstracts. We pick `ccdv/govreport-summarization`, `ccdv/pubmed-summarization`, `ccdv/arxiv-summarization`(Cohan et al., 2018), which includes different types of documents from arxiv articles to government reports.

Math: We pick OpenAI GSM8K (Cobbe et al., 2021), a classic dataset of grade-school math problems designed to evaluate multi-step reasoning. We choose not to use more complex math-reasoning datasets because our focus in this work is primarily on LLM text-generation tasks. Advanced math reasoning often requires specialized methodologies, such as tree-search reasoning, which makes it unsuitable for single-pass direct generation.

Conditional Generation: The LLM should generate coherent text according to given constraints. In our setting, we task the LLM with filling in missing paragraphs in an essay or producing a complete essay based on an abstract outline. We use `qwedsacf/ivypanda-essays` (Qwedsacf, 2019), a HuggingFace dataset repository containing long-form essays covering multiple disciplines sourced from the *IvyPanda platform*[3].

B.2 Data Filtering Prompt

```
You are given a set of task samples, each
    consisting of:
1. User Query  the task or request made to the
    model.
2. Model Response  the output given by the model.

The samples may come from various task types,
    including:
- Translation
- Summarization
- Math problem solving
- Reinforcement Learning from Human Feedback (
    RLHF) style instructional prompts
- Conditional text generation
- Multiturn dialogue

Your goal: Identify and select only the samples
    that did not meet
quality standards based on:

A. Query Quality Issues:
- Illformed or incomplete queries
- Ambiguous or misleading instructions
- Irrelevant or off-topic requests
- Grammatically broken or nonsensical input

B. Response Quality Issues:
- Incorrect or factually wrong answers
- Incomplete responses that fail to address the
    query
- Poor language quality or incoherent writing
- Hallucinations or madeup facts
- Misinterpretation of the query

Instructions:
1. For each sample, examine both the query and
    response.
2. Mark the sample as "Fail" if either the query
     quality
   or the response quality is below standard.
3. Briefly explain why the sample fails,
   citing issues in query, response, or both.
4. Output only the failing samples, in the
    format:
   [Sample ID]
   Query: ...
   Response: ...
   Fail Reason: ...

Be strict in applying the criteria  even if only
    one
side (query or response) is substandard, the
    sample
should be considered as failing.
```

C Experiment Training Details

C.1 Training Configuration

For the SFT setting, we tuned the base model on the training dataset for 2 epochs. All RL experiments

---

[3] https://ivypanda.com/

were conducted using the GRPO (Shao et al., 2024) algorithm framework for 80 steps in total. Since Qwen3-8B is a reasoning model, we enforced a formatting constraint: the reward was set to zero if the reasoning chain failed to be properly encapsulated within ‘<think>’ and ‘</think>’ tags. We did not apply a length penalty during RL training, as generation length does not consistently correlate with task performance. More training details are filed in Appendix C.3.

All experiments were performed on a cluster 8×NVIDIA H100 GPUs (80GB), using a global batch size of 128 and mixed-precision (FP16) training. An auxiliary cluster of 8×NVIDIA A100 GPUs (80GB) was dedicated to the deployment and inference of the reward models.

C.2 Prompt for the LLM Judge

search results filtration

Input: propmt, candiate, reference

You are an expert evaluator of language model outputs. You will receive:
1. **Prompt:** The original instruction/task given to the model.
2. **Candidate Response:** The model’s output to be evaluated.
3. **Reference Response:** A high-quality gold-standard or reference output.
Your task:
- Evaluate the quality of the *Candidate Response* compared to the *Reference Response* and in relation to the given *Prompt*.
- Consider the category of task (which could be: **translation**, **summarization**, **generation**, **infilling/cloze**, **conditional generation**, **math**, or **instruction following**), and adjust your evaluation criteria accordingly.
- Score on a scale from 0 to 10, according to the rubric below.
- Output the score in the format ‘[[X]]‘ (where X is the integer score) **once** in your reply, followed by a clear explanation of reasoning and specific strengths/weaknesses.
—
Evaluation Dimensions by Task Category *(Use whichever are relevant to the given prompt.)*
- **Translation:** Accuracy, completeness, fidelity to meaning, fluency, grammar, style.
- **Summarization:** Coverage of key points, factual faithfulness, conciseness, coherence.
- **Generation (creative writing, open-ended):** Relevance, originality, creativity, coherence, style, adherence to constraints.
- **Infilling/Cloze:** Correctness of missing content, contextual fit, fluency, logical continuity.
- **Conditional Generation:** Logical or rule-based conformity, adherence to provided constraints, completeness.
- **Math/Reasoning:** Correctness of calculations or logic, clarity, rigor of explanation.
- **Instruction Following:** How fully and correctly the instructions are followed, alignment with intent, completeness.
—
Scoring Rubric (0–10)
- 10: Perfect or near-perfect match. Fully correct, faithful, or relevant. No significant errors in meaning, facts, or execution. High clarity, fluency, and adherence to task.
- 9: Almost perfect; tiny, easily overlookable issues (minor style or formatting quirks).
- 8: Very good; only minor errors or slight omissions that don’t significantly harm the result.
- 7: Good; mostly correct but with notable small issues (minor factual, structural, or stylistic errors).
- 6: Fair; significant issues exist but main content or logic remains intact. Some loss of fidelity, clarity, or completeness.
- 5: Borderline acceptable; mix of correct and incorrect elements, noticeable gaps or errors, not reliably usable without fixes.
- 4: Poor; frequent errors or omissions, core meaning partially lost. Low reliability.
- 3: Very poor; large parts incorrect, irrelevant, or incoherent. Only minor parts are correct.
- 2: Minimal correctness; almost entirely wrong or off-task, but with a trace of relevant material.
- 1: Nearly useless; incomprehensible or

totally wrong, but not fully empty.
- 0: No meaningful output, completely unrelated, or empty.
—
Output Format Respond with: “‘ [[X]] Explanation: [Your detailed explanation, citing specific task-related criteria, success points, and failure points. Mention the type of category-specific evaluation applied.] “‘ - Replace **X** with a single integer 0–10. Make sure your explanation is concise within 50 words.
—
[propmt]
{prompt}
[Candidate Response]
{candidate}
[Reference Response]
{reference}

C.3 Hyperparameters Details for RL Training

We use the volcano engine reinforcement learning for LLMs framework, VERL (Sheng et al., 2024). We validate the implementation of the framework run all our RL experiments based on it. Below is the hyperparameters for all our experiments and we use the same set of hyperparameters for all experiments.

The other hyper-parameters, such as optimizer β, are set default to the framework trainer configurations from `https://github.com/volcengine/verl/tree/main/verl/trainer/config`.

Category / Parameter	**Value**
Training & Data	
Total Epochs	5
Global Train Batch Size	256
Max Prompt Length	10,000
Max Response Length	5,000
Truncation Strategy	’error’
Filter Overlong Prompts	True
Algorithm (GRPO/PPO)	
Advantage Estimator	GRPO
Learning Rate	1×10^{-6}
PPO Mini-batch Size	16
KL Coefficient	0.0001
KL Loss Type	low_var_kl
Entropy Coefficient	0
KL in Reward	False
Rollout (vLLM Engine)	
Rollout Samples (n)	8
Max Batched Tokens	65,536
GPU Memory Utilization	0.7
Prompt/Response Length	10k / 5k
Parallelism & Infrastructure	
Nodes / GPUs per Node	1 / 8
Tensor Parallel (TP) Size	2
Pipeline Parallel (PP) Size	4
Gradient Checkpointing	True
Micro Batch Size (Actor)	1

Table 4: Hyperparameter settings for GRPO training with Qwen3-8B.

D Prompt Details

D.1 Prompt for Task Decomposition

search results filtration

Input: original_task

Please break down the following generative task into a combination of several basic generative tasks:
Basic task list: 1. Controlled generation: Generate coherent natural language text that meets certain given conditions. Best for simple, clear tasks; complex writing should be split into smaller steps like planning and cloze generation.
2. Translation: Generate a corresponding text in another natural language from a text in one natural language.
3. Text summarization: Summarize the given text, retaining the main information.
4. Question answering: Provide appropriate answers based on background information and question requests provided by the user.
5. Paraphrasing: Modify the provided text into a different form of expression that meets the given rewriting requirements.
6. Cloze generation: Given a continuous piece of text with missing parts, generate appropriate text for the missing positions so that the original text becomes complete, coherent, and consistent.
7. Planning generation: Plan a high-level outline in order to accomplish a relatively complex generative task, such as creating a chapter list, designing character traits, designing scripts, or designing a timeline.
8. Code: Generate executable code that meets the specified requirements, or supplement or revise code according to the given requirements. The defining criterion for this task is that the output is primarily code.

Category / Parameter	Value
Training & Data	
Total Epochs	5
Global Train Batch Size	256
Max Prompt Length	10,000
Max Response Length	5,000
Truncation Strategy	'error'
Filter Overlong Prompts	True
Algorithm (GRPO)	
Advantage Estimator	GRPO
Learning Rate	1×10^{-6}
PPO Mini-batch Size	16
KL Coefficient	0.0001
KL Loss Type	low_var_kl
Entropy Coefficient	0
KL in Reward	False
Rollout (vLLM Engine)	
Rollout Samples (n)	8
Max Batched Tokens	65,536
GPU Memory Utilization	0.7
Prompt/Response Length	10k / 5k
Parallelism & Infrastructure	
Nodes / GPUs per Node	1 / 8
Tensor Parallel (TP) Size	2
Pipeline Parallel (PP) Size	4
Gradient Checkpointing	True
Micro Batch Size (Actor)	1

Table 5: Hyperparameter settings for GRPO training with Llama-3.1-8B.

Decomposition goal:
- Break down the complex generative task provided by the user into a list composed of the above basic tasks according to its logical steps. - Steps should be arranged in execution order, and the description should start from the original input form and proceed until the task is completed.
- Each step must clearly specify the "basic task type" and the execution content of that step.
- If the task does not need to be broken down, provide a single-step basic task and rewrite its description into a clearer instruction that aligns with the type of task in the basic task list.
Output format requirements:
- List the decomposition results step-by-step (step number + basic task type name + specific execution description).
- Enclose the final result within <Result> ... < \Result > tags.
Below is an example:
[Example Start]
Task to be decomposed: Please provide an English summary for the following Chinese document.
Decomposition result:
1. Translation: Please translate the following Chinese document into an English document.
2. Text summarization: Please summarize the given English document, and ensure the summary does not exceed 200 words.
[Example End]
Now, perform the above decomposition process on the given question (or task description) below, and write the final decomposition result within <Result> ... < \Result > tags.
{original_task}

D.2 Prompt Details for Reward Model Choice

D.2.1 Tool wrapping

```
{
    "type": "function",
    "function": {
        "name": "search_serper_engine",
        "description": "Performs a Google search
             using the Serper API restricted to
             finding Hugging Face model
             checkpoints. Use this tool only to
             look up Hugging Face checkpoint URLs,
              model pages, or related information.
              Short queries work best. Reward
             model might be confusing with base
             models or chat models",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "The search
                        query for Hugging Face
                        checkpoints, e.g., model
                        names or keywords to
                        locate on huggingface.co."
                }
            },
            "required": ["query"]
        }
    }
}
```

D.2.2 Prompt for search results filtration

search results filtration

Input: original_task

You are given a list of search engine results with position IDs. Your task is to filter them

according to the following rules:
1. **Identify Reward Models:**
- Keep only results that are **reward model** links.
- Reward models often have model names containing keywords like '-Reward-' or '-RM-'.
- Discard results for base models ('-Base') or instruct models ('-Instruct') or chat models ('-Chat').
- If a model name has none of these hints, and it's unclear whether it is a reward model, discard it.
2. **Hugging Face Model Repositories Only:**
- Keep only links pointing to **Hugging Face model repositories**.
- Discard datasets, research papers, blog posts, or other non-model content.
3. **Score Output Format only:**
- Regression models only, in other words, models that output a score (e.g., 0-1) rather than generating text.
Directly discard those items that violates rule 1, 2 or 3 and keep the rest items. Output the resting items in list using their original position id like "[0, 1, 3, 5, ...]". If none of the items are left, output an empty list "[]".
{results}

D.2.3 Prompt for search results Reranking

search results rerank

Input: original_task

You are given a list of search engine results with position IDs. Your task is to filter them according to the following rules:
1. **Identify Reward Models:** - Keep only results that are **reward model** links. - Reward models often have model names containing keywords like '-Reward-' or '-RM-'. - Discard results for base models ('-Base') or instruct models ('-Instruct') or chat models ('-Chat'). - If a model name has none of these hints, and it's unclear whether it is a reward model, discard it.
2. **Hugging Face Model Repositories Only:** - Keep only links pointing to **Hugging Face model repositories**. - Discard datasets, research papers, blog posts, or other non-model content.
3. **Score Output Format only:** - Regression models only, in other words, models that output a score (e.g., 0-1) rather than generating text.
Directly discard those items that violates rule 1, 2 or 3 and keep the rest items. Output the resting items in list using their original position id like "[0, 1, 3, 5, ...]". If none of the items are left, output an empty list "[]".
{results}

D.2.4 Prompt for search results LLM-based Reward Model Implementation

reward tool implementation

Input: original_task

Implement a python script for launching a reward model according to the following informative scripts. The model local checkpoint is {model_local_dir}. The cuda device for the model is "{cuda_device}". You should write a function, that support input parameter: - prompt: str, instruction or context conditions - response: str, the text need to be evaluated - reference: str, some reference answer/response for the above prompt Your implementation are free to use the packages mentioned in the scripts. Name the calculation function starting with "compute_", such as "def compute_XXX(...)" where XXX should be the reward model name or related abbreviation. Make sure the model checkpoint is loaded precisely once in the script. Format your output enclosed within "python \ n xxxx \n". Also, additionally print the calculation funciton after four sharp marks ####, such as "#### def compute_xxx(...)" in the end of your output (outside the python script).
{scripts}
[your implementation]

D.3 Prompt for Code-Agent workflow

D.3.1 Plan

```
LIST_TASK_PROMPT = """You are an expert in
    designing reward models and evaluation
    metrics for the **{task}** task.
```

```
Your goal is to list **35 possible reward model
    or evaluation metric choices** for this task
    , drawing from the following two categories:

1. **Rule-based**  Explicit rules (e.g., exact
    match with reference output, length
    constraints) used directly as rewards.
2. **Metric-based**  Standard NLP metrics (e.g.,
     BLEU, ROUGE, METEOR) used to evaluate and
    reward generated results.

**Output formatting requirements:**
- Place your results **after four hash marks
    (‘####‘)**.
- For **each choice**, indicate its **category**
     and **name**, using the format:
  ‘‘‘
  #### <Category>/<Name>: <Brief description>
  ‘‘‘
- Use a **new line** for each choice.

**Example:**
‘‘‘
#### Metric-based/BLEU: Measures the n-gram
    overlap between generated output and
    reference text.
#### Rule-based/Length: Rewards outputs within
     the target length range for conciseness.
‘‘‘
"""
```

D.3.2 Write

```
WRITE_CODE_PROPMT = """Implement the following
    metric according to description using python
    . You are free to use packages. You should
    write a function begin with 'compute_xxx'
    where xxx is the name of the metric. The
    function accepts:
- prompt: the instruction to the prompt
- candidate_response: the candidate response to
    be evaluated by the metric
- reference_response: the reference answer for
    the prompt
You should directly return a scaler score.

Output the python code in ‘‘‘python\n xxxx\n‘‘‘.
     And list the requirements within ‘‘‘‘‘‘ use
     requirements.txt style.

{metric description}
"""
```

D.4 Prompts for WrapLLM and CodeVerify

```

TASK_CLS_PROMPT = """
You will be given a ‘question‘ and an ‘answer‘
    from a language model interaction.
Your job is to determine the main type of task
    being performed. Examples include but are
    not limited to: translation, summarization,
    question answering, code generation, math
    solving, creative writing, RLHF alignment,
    explanation, classification, etc.
You do not need to list all possible task types
     instead, use your judgment to give the most
     fitting label for this specific instance.
Rules:
1. Output only the task type as a concise label
     (maximum **three words**).
2. Do not include any extra text, punctuation,
     or explanation.
3. If uncertain, choose the closest-fitting
     description.

Provide your output results after four sharp
    marks ####, such as "#### Translation".

**Input example:**
‘‘‘json
"question": "You are a helpful assistant that
    translates English sentences to French.
    Following the below input-output format\n[
    English Input]\nxxxx\n[French Output]\nxxxx\
    nStart your translation task now. [English
    Input]\nMichael Gill formed The Murder Mile
    with his old friend, ex- Spring Heeled Jack
    USA and Lost City Angels frontman, Ron
    Ragona.\n[French Output]\n",
"answer": "Michael Gill a form The Murder Mile
    avec son vieil ami, l’ancien chanteur de
    Spring Heeled Jack USA et Lost City Angels,
    Ron Ragona."
‘‘‘
**Output:**
#### EnglishFrench Translation

Now classify the given instance according to
    these rules.

"""


TASK_DECOMP_PROMPT = """Please break down the
    following generative task into a combination
     of several basic generative tasks:

Basic task list:
1. Controlled generation: Generate coherent
    natural language text that meets certain
    given conditions. Best for simple, clear
    tasks; complex writing should be split into
    smaller steps like planning and cloze
    generation.
2. Translation: Generate a corresponding text in
     another natural language from a text in one
     natural language.
3. Text summarization: Summarize the given text,
    retaining the main information.
4. Question answering: Provide appropriate
    answers based on background information and
    question requests provided by the user.
5. Paraphrasing: Modify the provided text into a
     different form of expression that meets the
     given rewriting requirements.
6. Cloze generation: Given a continuous piece of
     text with missing parts, generate
    appropriate text for the missing positions
    so that the original text becomes complete,
    coherent, and consistent.
7. Planning generation: Plan a high-level
    outline in order to accomplish a relatively
    complex generative task, such as creating a
    chapter list, designing character traits,
    designing scripts, or designing a timeline.
8. Code: Generate executable code that meets the
     specified requirements, or supplement or
```

```
    revise code according to the given
    requirements. The defining criterion for
    this task is that the output is primarily
    code.


Decomposition goal:
- Break down the complex generative task
    provided by the user into a list composed of
     the above basic tasks according to its
    logical steps.
- Steps should be arranged in execution order,
    and the description should start from the
    original input form and proceed until the
    task is completed.
- Each step must clearly specify the basic task
    type and the execution content of that step.
- If the task does not need to be broken down,
    provide a single-step basic task and rewrite
     its description into a clearer instruction
    that aligns with the type of task in the
    basic task list.


Output format requirements:
- List the decomposition results step-by-step (
    step number + basic task type name +
    specific execution description).
- Enclose the final result within ‘<Result> ...
    </Result>‘ tags.

Below is an example:
[Example Start]
Task to be decomposed: Please provide an English
     summary for the following Chinese document.
Decomposition result:
1. Translation: Please translate the following
    Chinese document into an English document.
2. Text summarization: Please summarize the
    given English document, and ensure the
    summary does not exceed 200 words.
[Example End]

**Now,** perform the above decomposition process
     on the given question (or task description)
     below, and write the final decomposition
    result within ‘<Result> ... </Result>‘ tags.


{original_task}
"""


LIST_TASK_PROMPT = """You are an expert in
    designing reward models and evaluation
    metrics for the **{task}** task.
Your goal is to list **35 possible reward model
    or evaluation metric choices** for this task
    , drawing from the following two categories:

1. **Rule-based**  Explicit rules (e.g., exact
    match with reference output, length
    constraints) used directly as rewards.
2. **Metric-based**  Standard NLP metrics (e.g.,
     BLEU, ROUGE, METEOR) used to evaluate and
    reward generated results.

**Output formatting requirements:**
- Place your results **after four hash marks (‘#
    ###‘)**.
- For **each choice**, indicate its **category**
     and **name**, using the format:
 ‘‘‘
 #### <Category>/<Name>: <Brief description>
 ‘‘‘
- Use a **new line** for each choice.

**Example:**
‘‘‘
#### Metric-based/BLEU: Measures the n-gram
    overlap between generated output and
    reference text.
#### Rule-based/Length: Rewards outputs within
    the target length range for conciseness.
‘‘‘

"""


WRITE_CODE_PROPMT = """Implement the following
    metric according to description using python
    . You are free to use packages. You should
    write a function begin with ’compute_xxx’
    where xxx is the name of the metric. The
    function accepts:
- prompt: the instruction to the prompt
- candidate_response: the candidate response to
    be evaluated by the metric
- reference_response: the reference answer for
    the prompt
You should directly return a scaler score.

Output the python code in ‘‘‘python\n xxxx\n‘‘‘.
     And list the requirements within ‘‘‘‘‘‘ use
     requirements.txt style.

[metric description]

"""


WRAP_TOOL_PROMPT = """Write a short description
    of the following reward function, based on
    the name, python code implementation. In
    your description, briefly explain what the
    function calculates and how it can be used
    to evaluate text generation quality. The
    description should be concise and clear,
    suitable for someone familiar with NLP
    evaluation metrics.

NAME
{name}

CODE

‘‘‘python
{code}
‘‘‘

write your description after four sharp marks #
    ###, such as "#### This function calculates
    ...".
"""



METRIC_DESIGN_PROMPT = """You are an expert in
    designing evaluation metrics writing with
    python code. Currently you nedd to implement
```

```
the {metric}. The introduction for the metric is as follows:

{info}

Suppose the input is prompt, candidate response, reference response. write the metric as a python function that receive the above parameter, and return with the score for it. Name it begin with compute_, such as def compute_XXX(...)

Make sure the code is correct and can be run without error. Include the script within ‘‘‘ python ... ‘‘‘.

"""


REWARD_MODEL_DESIGN_PROMPT = """You are an expert in launching reward models with python code. Write a script that supports calculating the reward score of some text. You should write a function, that support input parameter:

- prompt: str, instruction or context conditions

- response: str, the text need to be evaluated

- reference: str, some reference answer/response for the above prompt

You are given the following README.md of one reward model, and the local model checkpoint path. The cuda device for the model is cuda:0. Name the calculation function starting with "compute_", such as "def compute_XXX(...)". Make sure the model checkpoint is loaded precisely once in the script.

README.md:

{readme}

local model checkpoint path: {model_path}
"""


RERANK_PROMPT = """You are given a list of search engine results with position IDs.
Your task is to filter them according to the following rules:

1. **Identify Reward Models:**
   - Keep only results that are **reward model** links.
   - Reward models often have model names containing keywords like ‘-Reward-‘ or ‘-RM-‘.
   - Discard results for base models (‘-Base‘) or instruct models (‘-Instruct‘) or chat models (‘-Chat‘).
   - If a model name has none of these hints, and its unclear whether it is a reward model, discard it.

2. **Hugging Face Model Repositories Only:**
   - Keep only links pointing to **Hugging Face model repositories**.
   - Discard datasets, research papers, blog posts, or other non-model content.


3. **Score Output Format only:**
   - Regression models only, in other words, models that output a score (e.g., 0-1) rather than generating text.

Directly discard those items that violates rule 1, 2 or 3 and keep the rest items. Output the resting items in list using their original position id like "[0, 1, 3, 5, ...]". If none of the items are left, output an empty list "[]".

{results}
"""


LLMRM_IMPLEMENT_CODE = """Implement a python script for launching a reward model according to the following informative scripts. The model local checkpoint is {model_local_dir}. The cuda device for the model is "{cuda_device}". You should write a function, that support input parameter:
- prompt: str, instruction or context conditions
- response: str, the text need to be evaluated
- reference: str, some reference answer/response for the above prompt

Your implementation are free to use the packages mentioned in the scripts. Name the calculation function starting with "compute_", such as "def compute_XXX(...)" where XXX should be the reward model name or related abbreviation. Make sure the model checkpoint is loaded precisely once in the script. Format your output enclosed within ‘‘‘python\n xxxx\n‘‘‘. Also, additionally print the calculation funciton after four sharp marks ####, such as "#### def compute_xxx(...)" in the end of your output (outside the python script).

{scripts}

[your implementation]

"""


SELECT_INFORMATIVE_FILES = """Below are files from a Hugging Face model repository. Your task is to identify and list files that are likely to contain informative content about the model's implementation:

{file_list}

Directly output the selected file names in a list format like ["file1.py", "file2/README.md", ...]. If none of the files are informative, output an empty list "[]". Make sure to only output the list without any extra text or explanation.
"""
```

E LLM usage in this paper

Large Language Models (LLMs) were used in the preparation of this work as a general-purpose assistance tool. Specifically, LLMs were employed in the following ways:

- **Translation Assistance**: Converting expressions and sentences from the author's native language into English.
- **Language Polishing and Grammar Revision**: Improving clarity, fluency, and grammatical correctness of the text, and ensuring that phrasing is natural in academic English.
- **Draft Review and Critique**: Providing feedback on drafts, including identifying unclear passages, suggesting improvements in structure, and flagging potential ambiguities.

LLMs were not used for generating original research ideas, performing data analysis, or writing substantive technical content. All core research contributions, results, and argumentative structure were developed by the authors. The role of LLMs was limited to translation, linguistic polishing, and non-substantive editorial suggestions to improve presentation.

F Generated Tools

Table 6: A list of the generated reward function tool names by our code-agent.

Type	Metric	Type	Metric
rule_based	`Forbidden_Words`	rule_based	`Stepwise_Completeness`
rule_based	`Prompt_Adherence`	rule_based	`Length`
rule_based	`Numeric_Accuracy`	rule_based	`Exact_Template_Match`
rule_based	`Novelty_Penalty`	rule_based	`Contradiction_Detection`
rule_based	`Disallowed_Phrase_Penalty`	rule_based	`Exact_Output_Match`
rule_based	`No_Unsupported_Claims`	rule_based	`Reference_Match`
rule_based	`Exact_Answer_Match`	rule_based	`Named_Entity_Preservation`
rule_based	`Unit_Consistency`	rule_based	`Keyword_Presence`
rule_based	`Minimal_Edit_Distance`	rule_based	`Thesis_Inclusion`
rule_based	`Mandatory_Content_Inclusion`	rule_based	`Scientific_Claims_Match`
rule_based	`Pronounceability`	rule_based	`Position_Sensitivity`
rule_based	`Section_Coverage`	rule_based	`Entity_Presence`
rule_based	`Answer_Type_Match`	rule_based	`Stepwise_Correctness`
rule_based	`Terminology_Accuracy`	rule_based	`Diversity_Score`
rule_based	`Forbidden_Content`	rule_based	`Fact_Match`
rule_based	`Forbidden_Phrase_Detection`	rule_based	`No_Information_Leakage`
rule_based	`Annotation_Completeness`	rule_based	`Grammar_and_Spelling_Accuracy`
rule_based	`Clarity_Constraint`	rule_based	`Answer_Presence`
rule_based	`No_Overlap_with_Input`	rule_based	`No_Syntax_Errors`
rule_based	`Numeric_Tolerance`	rule_based	`Edit_Distance`
rule_based	`Keyword_Coverage`	rule_based	`No_Repetition`
rule_based	`Length_Ratio`	rule_based	`One-Hot_Accuracy`
rule_based	`Novelty`	rule_based	`Exact_Match`
rule_based	`Pattern_Compliance`	rule_based	`Step_Match`
rule_based	`Syntax_Validity`	rule_based	`Format_Compliance`
rule_based	`Allowed_Vocabulary`	rule_based	`Entity_Overlap`
rule_based	`Explicit_Irrelevance`	rule_based	`Accuracy`
rule_based	`Coverage_of_Key_Points`	rule_based	`Section_Presence`
rule_based	`Clarity`	rule_based	`Test_Case_Pass_Rate`
rule_based	`Dictionary_Filtering`	rule_based	`Length_Expansion`
rule_based	`Content_Inclusion`	rule_based	`Error_Pattern_Removal`
rule_based	`Plagiarism_Check`	rule_based	`Functionality_Test`
rule_based	`Politeness_Constraint`	rule_based	`Formatting_Compliance`
rule_based	`Exact_Test_Case_Pass`	rule_based	`Key_Information_Coverage`
rule_based	`Genre-Adherence`	rule_based	`Passes_Unit_Tests`
rule_based	`Exact_Step_Match`	rule_based	`Exact_Keyword_Match`
rule_based	`Required_Field_Inclusion`	rule_based	`Attribute_Coverage`
rule_based	`Valid_Vocabulary`	rule_based	`Medical_Term_Coverage`
rule_based	`Keyword_Absence`	rule_based	`Required_Component_Presence`
rule_based	`Final_Answer_Correctness`	rule_based	`Keyword_Inclusion`
rule_based	`Structure_Compliance`	rule_based	`Step_Consistency`
rule_based	`Readability`	rule_based	`No-Answer_Accuracy`
rule_based	`Length_Constraint`	rule_based	`Error_Reduction`
rule_based	`Answer_Type_Mismatch`	rule_based	`Thesis_Presence`
rule_based	`Case-Insensitive_Match`	rule_based	`Topic_Divergence`
rule_based	`Exact_Numeric_Match`	rule_based	`Originality-Penalty`
rule_based	`Keyword_Exclusion`	rule_based	`Structure`
rule_based	`Format_Consistency`	rule_based	`Required_Elements`
rule_based	`Reference_Citation`	rule_based	`Instruction_Match`

Type	**Metric**	**Type**	**Metric**
rule_based	`Key_Concepts_Inclusion`	rule_based	`Stepwise_Solution_Match`
rule_based	`Fact_Consistency`	rule_based	`Step_Count_Constraint`
nlp_metric	`F1_Score`	nlp_metric	`METEOR`
nlp_metric	`ROUGE`	nlp_metric	`GLEU`
nlp_metric	`BERTScore`	nlp_metric	`M^2_Score`
nlp_metric	`chrF`	nlp_metric	`ROUGE-L`
nlp_metric	`Levenshtein_Distance`	nlp_metric	`BLEU`
nlp_metric	`Distinct-n`	nlp_metric	`CodeBLEU`
model_based	`Content_Novelty_Score`	model_based	`Negative_Relevance_Score`
model_based	`Topic_Classifier`	model_based	`Perplexity`

Source	Repo Name
(Liu et al., 2024)	Skywork/Skywork-Reward-V2-Llama-3.1-8B
(Liu et al., 2024)	Skywork/Skywork-Reward-V2-Qwen3-8B
(Liu et al., 2024)	Skywork/Skywork-Reward-V2-Llama-3.2-3B
(Liu et al., 2024)	Skywork/Skywork-Reward-V2-Qwen3-4B
(Cheng et al., 2025)	ByteDance-Seed/Seed-X-RM-7B
	OpenAssistant/reward-model-deberta-v3-base
(Yang et al., 2024)	Ray2333/gpt2-large-helpful-reward_model
(Corrêa, 2023)	nicholasKluge/RewardModel
(Cai et al., 2024)	internlm/internlm2-1_8b-reward
(Malik et al., 2025b)	allenai/Llama-3.1-8B-Base-RM-RB2

Table 7: Successfully Top@9 deployed LLM-based reward models.

Task Type	Unmatched(%)
Infilling	47.4
Essay Generation	43.8
Multi-Turn	8.8

Table 8: Web Retrieval Page Rank

G Additional Analysis

G.1 Error and Robustness Analysis

We conducted an error analysis by counting the task types of instructions for which the Web-Agent could not find a specialized reward model (unmatched conditions). The breakdown in Table 9 shows that the majority of unfound instructions originate from essay infilling/generation tasks. Specifically, there is currently no corresponding reward model explicitly trained for the these two task domains, which accounts for the high unmatched ratio in these categories. Notably, When a specialized tool is unmatched, RLAR defaults to using a generic, default LLM-based reward model (`skywork-llama`).

To assess the robustness of the searching module, we tracked the average item position (calculated as page rank ×10) for the matched reward model. Across all sampled categories, the overall average retrieval position was 5.64 items. As detailed in Table 8, all individual sub-categories consistently found the optimal item on the first page, confirming the robustness and high precision of the agent's query generation and search logic.

We further validate the soundness of the framework's design by including a detailed analysis of the generated tool quality (Appendix G.2) and an investigation into the reward tool usage within our main experiments (Appendix G.3). In summary, code-agents achieve a 94.9% executable rate when utilizing rule/metric-based tools, and we observe a dominant percentage of LLM-based reward tool usage in text-generation tasks.

Category	Avg Pos
Summ	7.17
Translation	2.36
RLHF	5.03
Multi-Turn	7.61
Infill/Gen	3.75
Math	6.87

Table 9: Position Ranks

Category	Count
Code-agent scripts (total)	118
Executable	112 (94.9%)
Rule-based	102 (86.4%)
Metric-based	12 (10.2%)
Learned-model–based	4 (3.4%)
Web-agent repos (retrieved)	21
Deployed	10 (47.6%)
Rejected (size)	2
Rejected (not classification)	6
Rejected (insufficient docs)	3

Table 10: Summary of reward tool generation outcomes.

G.2 Reward Tool Generation Quality

We evaluate the quality of reward tools produced by our two agents for tool generation mainly along their construction validity and summarized in Table 10.

Code-agent tools. Across all the training queries, the code agent generated **118** reward scripts, among which **112** (**94.9%**) were directly executable under our standardized interface.[4] By type, the set comprises **102** rule-based functions (**86.4%**), **12** standard metric implementations (**10.2%**; e.g., BLEU, METEOR), and **4** learned-model–based scorers (**3.4%**). Rule-based tools typically encode task-specific verifiable criteria (e.g., numeric-consistency checks for GSM8K or explicit-irrelevance penalties for RLHF-style preference items), while metric-based tools provide length- or n-gram–aware surrogates for general text quality. We discard learned-model–based proposals from the code agent since they are potential out of memory threats to the deploying server.

Web-agent tools. The web agent retrieved **21** candidate repositories from public model hubs (primarily Hugging Face and ModelScope) that matched the predicted task label and satisfied our reward-model filter. And the filter eliminates

---

[4] Executability is checked by importing the generated function, calling it with a minimal synthetic triplet (`prompt, candidate, reference`) and verifying a numeric return type without exceptions.

base/instruct/chat/vision models and retains text-classification modeled reward models with download access. After automatic screening and wrapping, **10** repositories (**47.6%**) were successfully deployed behind a uniform Python API. The remaining **11** were rejected due to: model size prohibitive for our inference node (**2**), non–text-classification architectures (**6**), or insufficient/ambiguous repository documentation for reliable wrapping (**3**).

The high executability of code-agent tools (**94.9%**) and the moderate but reliable deployment rate of web-agent tools (**47.6%**) indicate that RLAR can consistently materialize task-aligned reward functions across heterogeneous inputs.

G.3 Reward Tool Usage and Selection Patterns

Having established that RLAR can reliably generate and deploy reward tools, we next examine how these tools are actually invoked during training. This analysis addresses two questions: (i) which categories of tools dominate in practice, and (ii) how the usage patterns vary with task source and affect the learned policy.

We plot the actual usage of tools by examining the tool matching conditions based on the data source in the training set shown in the Figure 4. Across all 8,000+ training samples, the majority of calls are routed to **LLM- based reward models** (**96.4%**), while **rule- based** and **metric-based** tools are invoked only sparsely. The most frequently selected individual model is `Skywork/Skywork-Reward-V2-Llama-3.1-8B`, accounting for **52.5%** of calls. A significant proportion of samples fall back to rule-based numeric-consistency checks ("explicit number match") On translation tasjs, the web-agent originated `Seed-X-RM-7B` dominates, capturing cross-lingual adequacy more effectively than generic reward models.

The dominance of LLM-based rewards suggests that, for heterogeneous open-domain training, high-capacity discriminative models remain the most trusted. Nevertheless, the occasional use of rule-based checks in math and RLHF tasks demonstrates that RLAR is capable of combining expert heuristics when appropriate. RLAR does not rely on a single global reward model but instead orchestrates a portfolio of evaluators aligned with each domain. As shown in the previous subsection, this diversity translates into smoother advantage estimation and stronger updates during policy optimization.

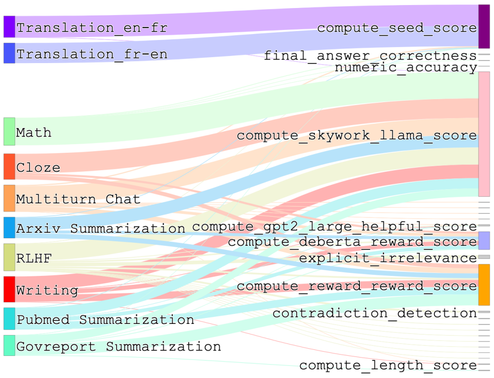


Figure 4: Matching tools with source training dataset distribution.

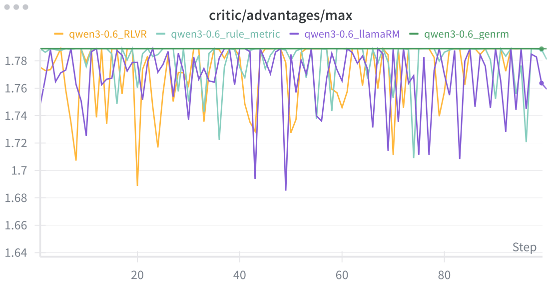


Figure 5: Maximum Advantage Estimations

G.4 Impact on Advantage Estimation and Policy Learning

We examine the records from the Qwen experiments covering Generative RM, method, single generic reward, regarding the estimated min/max of advantage per step (Figure 5 and Figure 6), and calculated the proportion that triggered clipping. Higher rates of being clipped means a higher absolute value of estimated advantage. From the results, for Generative RM, rollouts triggering both upper-clip and under-clip occur in every update step. Compared to single generic reward, RLAR has a significantly higher clipping rate. This is direct evidence that **methods with better performance tends to estimate larger advantages in absolute values**.

Return to the discussion of Advantage Estimation $\hat{A}_i = \frac{r_i - \text{mean}(r)}{\text{std}(r)}$. Consider two types of reward functions in Figure 7, the blue one is sensitive to extreme values (smaller variance) while the orange one is evenly modeled (higher variance). Assuming uniform roll-out sampling, a higher value of $\hat{A}_i$ suggests that the underlying reward function resembles **the sensitive type** (blue line). Therefore,

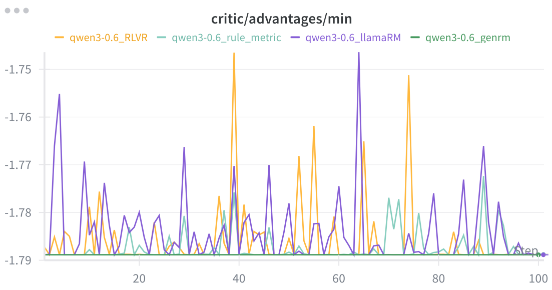


Figure 6: Minimum Advantage Estimations

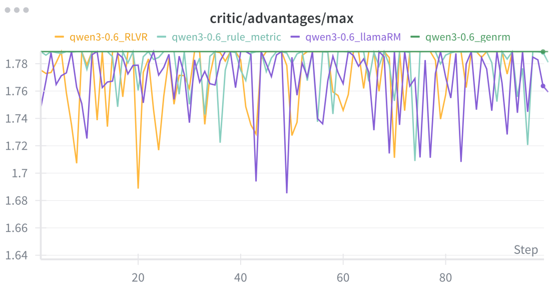


Figure 7: An illustration on the sensitivity to extreme values of reward functions.

extreme values (maximum/minimum) are divided by a smaller variance, resulting in a more frequent reaching of the clip threshold. This is expected for policy optimization that more weights should be transferred to these deviated rolls, as part of exploration-exploitation balance.