Improving LLM-Generated Code Quality with GRPO

Anonymous authors
Paper under double-blind review

Keywords: LLMs, code generation, code quality

Summary

Automated coding is a key focus in developing modern LLMs. Typically, these approaches use execution feedback as a reward signal, typically whether the generated code passes a number of tests. However, this reward signal has no notion of code quality, and indeed, the quality of generated code is a common complaint from experienced software engineers. We develop a comprehensive library `cisq_analyzer` to quantify code quality, and use it as a reward in GRPO. We find GRPO increases code quality according to this measure, which is confirmed by expert, blinded human annotators.

Contribution(s)

1. A comprehensive library, `cisq_analyzer` which captures notions of code-quality as defined by CISQ, mapping issues back to CWE IDs, and giving scores suitable for use within an RL pipeline.
   Context: None
2. Demonstrating that GRPO with a code-quality reward can indeed improve the quality of generated code, as evaluated by expert human annotators.
   Context: None

Improving LLM-Generated Code Quality with GRPO

Anonymous authors
Paper under double-blind review

Abstract

Large Language Models (LLMs) are gaining widespread use for code generation. Recent training procedures use execution feedback as a reward signal, typically focusing on the functional correctness of the code, using unit test pass rate as a reward signal. However, this reward signal fails to capture notions of maintainability, quality and safety of the code produced. We address this under-explored area and develop a comprehensive library to quantify various aspects of code quality, and use it as a reward in GRPO. We find GRPO increases code quality according to this measure, which is confirmed by expert, blinded human annotators.

1 Introduction

An increasing proportion of the world's software is being generated by LLMs. However, if LLM code generation is to continue gaining trust and adoption, understanding and improving the quality of LLM generated code is essential: it is quite possible for a "vibe-coded" software component to become unmaintainable by LLMs or expert software engineers, be open to critical security vulnerabilities, or waste vast amounts of energy by e.g. using a quadratic algorithm where a linear one is available.

Recently, these LLM coding systems have been trained using *execution feedback* as a reward signal (e.g. Shojaee et al., 2023; Dou et al., 2024; Ye et al., 2025; Dai et al., 2025; Gehring et al., 2025; Yang et al., 2024; Sorokin et al., 2025; Le et al., 2022; Liu et al., 2024). An example of an execution feedback setting is to take a coding specification or question (e.g. "write a function that generates Fibonacci numbers") paired with a number of tests. After the LLM generates code to solve the problem, we check whether that code passes the tests. If the code does pass the tests, then that is counted as a correct response and rewarded e.g. in a GRPO (Shao et al., 2024) pipeline. If the code does not pass one or more of the tests, then that is counted as an incorrect response and is penalized.

However, these rewards lack any notion of software quality, including:

- Maintainability: how easy is the software to keep working with and modify in the future
- Security: how well protected the code is against vulnerabilities
- Reliability: how effectively the software ensures availability, fault tolerance, and recoverability
- Performance: how well the generated code uses resources

In order to improve the usefulness and adoption of LLM generated code, we must look beyond merely rewarding functional correctness, but include incentives to produce code with good style and quality attributes, which make the code more readable, easier to work with and extend.

We thus sought to introduce a new family of rewards for e.g. GRPO that care about code-quality. That of course required us to programmatically quantify code-quality. Thankfully, quantifying code-quality is an area that has been well studied in the Computer Science literature (e.g. McCabe, 1976; Halstead, 1977; Chidamber & Kemerer, 1994; Fowler, 1999; Martin, 2008; Lanza & Marinescu, 2007), so there are many reasonable automated metrics which capture many of the four broad aspects listed above. As a concrete starting point, considered the list of automated source code quality measures[1] from the Consortium for Information & Software Quality (CISQ) (CISQ, 2025). We developed a Python library, `cisq_analyzer`, which implements analyzers for many of the common code weaknesses identified in the CISQ standards, and in turn evaluate the quality of Python code. This includes the ability to map identified code flaws to their Common Weakness Enumeration (CWE) ID, and return scalar code quality scores suitable for use within an RL pipeline. Next, we investigated whether incorporating this reward in a GRPO pipeline would improve code-quality in practice, relative to a control GRPO pipeline with no code-quality reward. We found that it did, both as measured by our code-quality metric, and by human annotators who were presented with answers from each of the resulting models, and asked to pick the one with better code quality. Of course, these annotators were blinded in the sense that the were not told which model each answer came from. Importantly, we also found that the model trained with a code-quality reward both performed as well or even better than the baseline model (measured using correctness, i.e. whether the code passes the tests), while also producing code of a shorter length on average than the baseline model. This means that at deployment-time, this intervention yields improved code quality while incurring no additional generation costs.

Our contributions are:

1. A comprehensive library, `cisq_analyzer` which captures notions of code-quality as defined by CISQ, mapping issues back to CWE IDs, and giving scores suitable for use within an RL pipeline.
2. Demonstrating that GRPO with a code-quality reward can indeed improve the quality of generated code, as evaluated by expert human annotators.

2 Related Work

There are a large number of papers using execution feedback to train LLMs to write code that passes tests (e.g. Shojaee et al., 2023; Dou et al., 2024; Ye et al., 2025; Dai et al., 2025; Gehring et al., 2025; Yang et al., 2024; Sorokin et al., 2025; Le et al., 2022; Liu et al., 2024). However, to our knowledge there is as of yet no work that combines this approach with reward terms to encourage improved code quality (our key contribution in this paper).

At the same time, there is a classical literature in computer science on code quality, including how to understand, improve and quantify it (e.g. McCabe, 1976; Halstead, 1977; Chidamber & Kemerer, 1994; Fowler, 1999; Martin, 2008; Lanza & Marinescu, 2007). However, to our knowledge there is as of yet no work that takes these metrics for code quality and uses them as a reward signal for training LLMs to produce higher-quality code.

Our `cisq_analyzer` library for assessing code-quality is designed to follow CISQ (2025), and uses a number of pre-existing libraries (The Pylint Team; Lacchia et al., a; Seipp et al.; Python Code Quality Authority; PyUp.io; The Mypy Team), along with a considerable number of "analyzers" written from scratch (see Sec. 3.1 for details). Importantly, these libraries have not, to our knowledge, been integrated into a comprehensive framework which produces a single number suitable for use as a reward in RL.

[1] https://www.it-cisq.org/cisq-files/pdf/cisq-weaknesses-in-ascqm.pdf

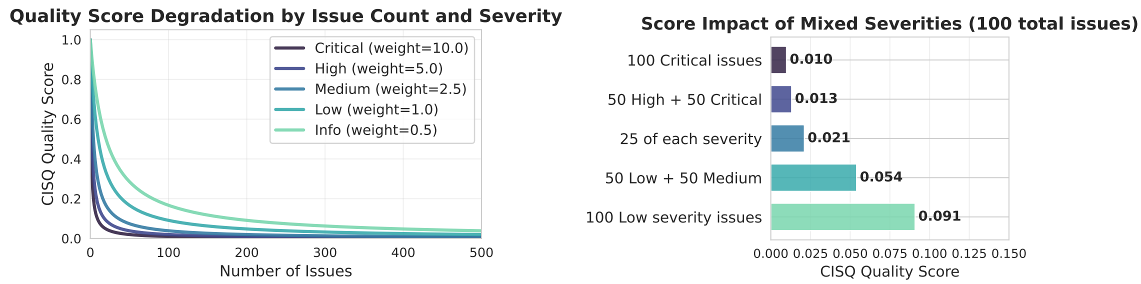

Figure 1: Code quality score evolution over number of issues and issue severity level

3 Methods

We use GRPO to improve the coding ability of various open-source LLMs. Such a pipeline involves making multiple choices, including the dataset and reward design, which we describe below.

3.1 Measuring Code Quality in RL Pipelines

We began by implementing a comprehensive library for evaluating the quality of Python code. We started with the CISQ Standards (CISQ, 2025). While comprehensive, these standards are in natural-language form, so not come with an official implementation. As such, we combined a number of existing libraries, such as Pylint (The Pylint Team), Radon (Lacchia et al., a), MyPy (The Mypy Team) and others that are able to detect code quality issues, while writing from-scratch a number of "analyzers" to detect issues that are missed by these general tools. We taxonomize the aspects of code quality that our analyzer picks up on in Table 1.

We collect these different code quality heuristics into a single `cisq_analyzer` library. Given a path to a directory of code to analyze, the main analysis function runs all the analysers for all characteristic groups (maintainability, security, performance, reliability) in parallel on the code, accumulating any found issues. These findings usually include a mapping to the CISQ CWE ID for categorization, and also include an assessment of the issue's severity into the set $\mathcal{S} = \{\text{info}, \text{low}, \text{medium}, \text{high}, \text{critical}\}$.

To obtain a numerical score which to train a model, we aggregate the findings as follows. First, we define the following weightings for each severity level, reflecting the relative importance to place on each type of issue identified:

$$w_{\text{info}} = 0.5, \quad w_{\text{low}} = 1.0, \quad w_{\text{medium}} = 2.5, \quad w_{\text{high}} = 5.0, \quad w_{\text{critical}} = 10.0.$$

Then, we let N_s be the number of findings at severity level s and calculate the weighted sum across severity levels

$$W = \sum_{s \in \mathcal{S}} w_s \, N_s,$$

following which we obtain a score between 0 and 1 using the following formula that decays with the weighted finding count, which is visually illustrated in Figure 1:

$$r_{\text{quality}} = \frac{1}{1 + W}.$$

Table 1: Taxonomy of Code Quality Issues Detected by CISQ Analyzer

Category	Example Issues	Analyzer
Maintainability		
Code Complexity	Excessive cyclomatic complexity Functions with high complexity scores Classes with overly complex methods	Radon (Lacchia et al., a) Xenon (Lacchia et al., b)
Dead Code	Unused functions, methods, and variables Unused class definitions and imports	Vulture (Seipp et al.)
Code Structure	Excessive function arguments Too many instance attributes Large files (>1000 LOC) Excessive branches/returns	Pylint (The Pylint Team)
Style & Documentation	Missing docstrings Poor naming conventions	Pylint
Security		
Code Injection	Shell injection (os.system) Unsafe subprocess calls Command injection risks	Bandit (Python Code Quality Authority)
Unsafe Data Handling	Insecure deserialization (pickle, YAML) Insecure XML parsing	Bandit
Cryptography	Weak hash algorithms (MD5, SHA1) Insecure random generation Hard-coded secrets	Bandit
Dependencies	Known vulnerable packages	*custom*
Performance		
String Operations	String concatenation in loops (+, +=)	*custom*
Resource Utilization	Resource-intensive loop operations Growing data structures in loops Network/file I/O in loops	*custom*
Data Structures	Excessive class attributes Deeply nested structures Large dictionaries	*custom*
Reliability		
Exception Handling	Bare/empty except clauses Overly broad exception catching Missing resource cleanup	*custom*
Concurrency	Lock ordering issues Missing lock releases	*custom*
Infinite Loops	Missing exit conditions Unchanging loop counters While True without breaks	*custom*
Type Safety	Type inconsistencies Missing annotations Incorrect argument/return types	Mypy (The Mypy Team)

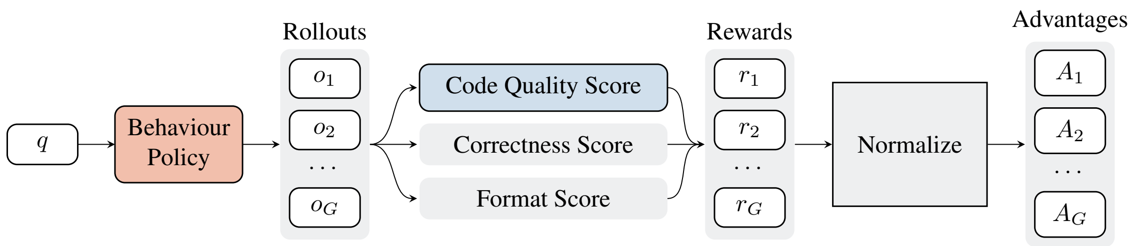

Figure 2: GRPO advantage calculation. In our experiments, we ablate the code quality score to quantify the benefit of including it.

3.2 Policy Optimization Algorithm

For completeness, we describe the Group Relative Policy Optimization (Shao et al., 2024) algorithm we use to train the model, and modifications from subsequent papers. The core idea behind GRPO is to sample multiple candidate outputs for a given query, and use their relative rewards to estimate advantages for policy updates. For each query sampled from the dataset $q \sim P(Q)$, GRPO samples a group G of outputs $o_1, \ldots, o_G$ using a behaviour policy $\pi_{\theta_{\text{old}}}$, corresponding to a previous iteration of the policy model, and updated periodically. The policy π_θ is updated by maximizing the following objective:

$$\begin{aligned} \mathcal{J}_{\text{GRPO}}(\theta) = {} & \mathbb{E}_{q \sim P(Q), \{o_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(O|q)} \\ & \frac{1}{G} \sum_{i=1}^{G} \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \min \left(\left[c_{i,t}(\theta) \hat{A}_{i,t},\ \text{clip}\big(c_{i,t}(\theta), 1 - \epsilon_{\text{low}}, 1 + \epsilon_{\text{high}}\big) \hat{A}_{i,t} \right] - \beta D_{\text{KL}} \left[\pi_\theta \| \pi_{\text{ref}} \right] \right), \end{aligned} \tag{1}$$

where

$$c_{i,t}(\theta) = \frac{\pi_\theta(o_{i,t}|q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q, o_{i,<t})}, \quad D_{\text{KL}} \left[\pi_\theta | \pi_{\text{ref}} \right] = \frac{\pi_{\text{ref}}(o_{i,t}|q, o_{i,<t})}{\pi_\theta(o_{i,t}|q, o_{i,<t})} - \log \frac{\pi_{\text{ref}}(o_{i,t}|q, o_{i,<t})}{\pi_\theta(o_{i,t}|q, o_{i<t})} - 1,$$

with clipping hyperparameters set to $\epsilon_{\text{low}} = 0.2$, $\epsilon_{\text{high}} = 0.28$ following (Yu et al., 2025), KL penalty strength coefficient $\beta = 0.001$ controlling stability and exploration (Schulman et al., 2017), and $\hat{A}_{i,t}$ being the group-relative advantage replicated for each token in the trajectory o_i

$$\hat{A}_{i,t} = \frac{r_i - \text{mean}(\{r_1, r_2, \ldots, r_G\})}{\text{std}(\{r_1, r_2, \ldots, r_G\})}.$$

We illustrate this in Figure 2.

3.3 Reward Design for Coding Tasks

Our rewards for each rollout r_i combines three components: a very simple format reward $r_{i,\text{format}}$ (which ensures the code can be parsed correctly), a code correctness reward $r_{i,\text{correct}}$ (which ensures the code functions correctly) as well as our code quality reward signal $r_{i,\text{quality}}$. Each of these range from 0 to 1.

Figure 3: Code quality score evolution over number of issues and issue severity level

The rewards for each rollout are linearly combined, with a slight emphasis on the code quality over the other components:

$$r_i = \frac{2}{10} r_{i,\text{format}} + \frac{3}{10} r_{i,\text{correct}} + \frac{5}{10} r_{i,\text{quality}}. \tag{2}$$

Format Reward Recent work has raised concerns that much of the performance improvement in RL on LLMs for coding can be explained by the LLM learning to generate code in the context of the specific prompts and tool format used, without much change to the code generation properties of the model itself (Shao et al., 2025; Chandak et al., 2025). To avoid this conflation and focus on the code quality, we simply prompt the model to output the solution in a Python markdown code block. We reward the model for one well formatted code block, and penalise multiple or incomplete code blocks.

Correctness Reward. The continuous correctness reward measures the held-out unit test pass rate, measuring whether the final code is functional and correct. A score of 0.0 indicates no tests passed, and 1.0 indicates all tests passed.

Code Quality Reward. This is the average score produced by `cisq_analyzer`. This ranges from 0 to 1, and does not rely on any 'labels' (i.e. unit test suite).

3.4 Synthetic Dataset Generation

Standard benchmarks like MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021), and APPS (Hendrycks et al., 2021) proved to be of limited utility for our analysis, as we found that the problems lacked the complexity required to meaningfully differentiate between solutions using established code quality metrics. The types of problems in these datasets are relatively self-contained and algorithmic in nature, and do not provide sufficient problem variation to demonstrate a number of code quality issues like credential handling, error handling, safe use of subprocesses and so on. Thus, we designed a synthetic data generation pipeline, which allowed us to generate multiple unique code-editing problems which were likely to highlight code quality issues in the models, while also being able to vary the theme, complexity and token length of the problems through prompting and filtering.

The synthetic data generation process is as follows: we first choose a problem category and a subcategory from lists spanning e.g. algorithm optimisation, data structures, programming paradigms, error handling and

Table 2: Reward components measured on the final iteration on the held-out validation problems.

Model	Validation Quality	Validation Correctness	Total Reward
Qwen 2.5 3B-Instruct	0.766	0.567	0.603
+ *quality reward training*	**0.878**(+**0.112**)	**0.690**(+**0.123**)	**0.785**(+**0.182**)
Llama 3.2 3B	0.859	0.601	0.627
+ *quality reward training*	**0.894**(+**0.035**)	**0.609**(+**0.008**)	**0.760**(+**0.133**)
OLMo 2 0425 1B Instruct	0.791	0.305	0.410
+ *quality reward training*	**0.864**(+**0.073**)	0.217(−0.088)	**0.631**(+**0.221**)

so forth. We then give Gemini 2.5 Pro (03-25) the category and subcategory, and prompt it to generate a problem statement. Before proceeding, we assess the conceptual novelty of the problem given the list of previously generated problems, and reject ones which are merely variations on previous problems, ensuring diversity. We then generate some starter code (e.g. a suboptimal solution), an ideal solution, and set of test cases. We finally iterate on the tests to ensure they are correct and pass with the reference solution.

See Appendix A for more detail about the dataset problems.

4 Results

We applied GRPO on Llama 3.2 3B Instruct (Team, 2024), Qwen2.5 3B Instruct (Qwen et al., 2025) and Olmo 2 1B Instruct (OLMo et al., 2025) with a dataset of 200 Python coding problems generated from our synthetic data generation pipeline, and our set of reward signals. We report the main findings in Table 2.

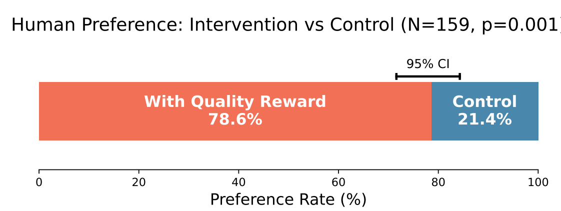

Figure 4: Human preferences of Qwen 2.5 3B output with and without the quality reward signal.

First, we found that models trained with and without the quality reward component performed similarly in terms of code correctness on the held-out set of validation coding problems, with slight improvements even observed in the Qwen and Llama models. When considering the quality reward component, as we might expect, the models trained with the `cisq_analyzer`-based quality reward component had higher code quality scores when evaluated on the held-out validation set.

To check for reward hacking, we presented human evaluators with pairs of solutions to validation problems generated from the models trained with and without the quality reward. These solutions were anonymized, presented in a random order, and the annotators were simply told to "*Choose which code snippet you think is of higher quality by clicking on it.*"

The human evaluators preferred the output from the model trained with the code quality component in 78.6% of 159 comparisons (95% CI: 71.6%-84.3%, $p < 0.001$, binomial test). The effect size was large (Cohen's h = 0.609). The position randomization was also effective (p = 0.812), with no significant position bias observed (p = 0.096).

Finally here is a short example of the difference in code produced in Figure 5. See Appendix B for some more examples. Despite this being a short problem, we can see the version from the model trained with the quality reward signal does not include the unused `math` import, includes type hints on the function, signature, and checks for an early return condition which simplifies the sum and index error handling later.

Without Quality Reward

```python
import math

def solution(data):
    n = len(data)
    fib_indices_to_sum = set()
    a, b = 0, 1
    while a < n:
        fib_indices_to_sum.add(a)
        a, b = b, a + b
    current_sum = 0
    for index in fib_indices_to_sum:
        try:
            current_sum += data[index]
        except IndexError:
            continue
    return current_sum
```

With Quality Reward

```python
def solution(data: list[int]) -> int:
    if not data:
        return 0

    n = len(data)
    a, b = 0, 1
    fib_indices = set()

    while a < n:
        fib_indices.add(a)
        a, b = b, a + b

    current_sum = sum(
        data[i] for i in fib_indices
    )

    return current_sum
```

Figure 5: Example code from models trained with and without the quality score.

The `cisq_analyzer` library is relatively CPU inexpensive, and executes all the analyzers in parallel to return the quality reward score in well under 1s per rollout.

5 Conclusions

In this work, we addressed the prevalent challenge of suboptimal code quality in Large Language Models (LLMs), which often stems from training methodologies that prioritize execution feedback over quality considerations. We introduced `cisq_analyzer`, a novel, comprehensive library grounded in CISQ standards, designed to quantify multiple facets of code quality—including maintainability, security, reliability, and performance—and translate them into a reward signal suitable for Reinforcement Learning (RL) pipelines. By incorporating this quality metric into a GRPO framework alongside rewards for correctness, and utilizing a purpose-built synthetic dataset reflecting real-world code-editing scenarios, we successfully trained LLMs to generate higher-quality code. Our findings indicate a significant improvement in code quality, as measured by our automated metrics and, importantly, validated by blinded expert human annotators without any degradation in the functional correctness of the generated code compared to baseline models.

References

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

Nikhil Chandak, Shashwat Goel, and Ameya Prabhu. Incorrect Baseline Evaluations Call into Question Recent LLM-RL Claims, May 2025. URL https://safe-lip-9a8.notion.site/Incorrect-Baseline-Evaluations-Call-into-Question-Recent-LLM-RL-Claims-2012f1fbf0ee8

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Carr, Michael J. Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nickolas Tezak, Sławek Stasiewicz, Ben Chess, Mohammad Bavarian, Kanishk Gandhi, Felipe Petroski Such, Jakub Pachocki, Sam A. Clune, John Schulman, Lukasz Kaiser, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

CISQ. Iso5055, 2025. URL https://www.it-cisq.org/standards/.

Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. Process supervision-guided policy optimization for code generation. *arXiv preprint arXiv:2410.17621*, 2025. Accepted to ICLR 2025.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. StepCoder: Improve Code Generation with Reinforcement Learning from Compiler Feedback. *arXiv preprint arXiv:2402.01391*, 2024.

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=zPPy79qKWe. arXiv preprint arXiv:2410.02089.

Maurice H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*, 2021.

Michele Lacchia et al. Radon: Code metrics in python, a. URL https://github.com/rubik/radon.

Michele Lacchia et al. Xenon: Python code complexity, b.

Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Science & Business Media, 2007.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C.H. Hoi. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. *arXiv preprint arXiv:2207.01780*, 2022.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Yang Wei, and Deheng Ye. RLTF: Reinforcement Learning from Unit Test Feedback. *Transactions on Machine Learning Research (TMLR)*, 2024. URL https://openreview.net/forum?id=hjYmsV6nXZ.

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, Michal Guerquin, Hamish Ivison, Pang Wei Koh, Jiacheng Liu, Saumya Malik, William Merrill, Lester James V. Miranda, Jacob Morrison, Tyler Murray, Crystal Nam, Valentina Pyatkin, Aman Rangapur, Michael Schmitz, Sam Skjonsberg, David Wadden, Christopher Wilhelm, Michael Wilson, Luke Zettlemoyer, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. 2 OLMo 2 Furious, January 2025. URL http://arxiv.org/abs/2501.00656.

Python Code Quality Authority. Bandit: A tool designed to find common security issues in python code. URL https://github.com/PyCQA/bandit.

PyUp.io. Safety: Check your python dependencies for known security vulnerabilities. URL https://github.com/pyupio/safety.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 Technical Report, January 2025. URL http://arxiv.org/abs/2412.15115.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, August 2017. URL http://arxiv.org/abs/1707.06347.

Jendrik Seipp et al. Vulture: Find dead python code. URL https://github.com/jendrikseipp/vulture.

Rulin Shao, Shuyue Stella Li, Rui Xin, Scott Geng, Yiping Wang, Sewoong Oh, Simon Shaolei Du, Nathan Lambert, Sewon Min, Ranjay Krishna, Yulia Tsvetkov, Hannaneh Hajishirzi, Pang Wei Koh, and Luke Zettlemoyer. Spurious rewards: Rethinking training signals in rlvr. https://rethink-rlvr.notion.site/Spurious-Rewards-Rethinking-Training-Signals-in-RLVR-1f4df34dac1880948858f95aeb88872f 2025. Notion Blog.

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

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

Nikita Sorokin, Ivan Sedykh, and Valentin Malykh. Iterative self-training for code generation via reinforced re-ranking. *arXiv preprint arXiv:2504.09643*, 2025. Published at ECIR 2025.

Llama 3 Team. The Llama 3 Herd of Models, November 2024. URL http://arxiv.org/abs/2407.21783.

The Mypy Team. Mypy: Optional static typing for python. URL https://github.com/python/mypy.

The Pylint Team. Pylint: code analysis for python. Webpage. URL https://pylint.org/.

Chengran Yang, Hong Jin Kang, Jieke Shi, and David Lo. ACECode: A Reinforcement Learning Framework for Aligning Code Efficiency and Correctness in Code Language Models. *arXiv preprint arXiv:2412.17264*, 1(1):20, December 2024.

Yufan Ye, Ting Zhang, Wenbin Jiang, and Hua Huang. PRLCoder: Leveraging Process-Supervised Reinforcement Learning to Enhance Code Generation. *arXiv preprint arXiv:2502.01715*, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An Open-Source LLM Reinforcement Learning System at Scale, May 2025. URL http://arxiv.org/abs/2503.14476.

A Synthetic Code Problem Examples

Our dataset generation is structured across the following problem categories: 'algorithm selection', 'array manipulation', 'custom structures', 'data structure choice', 'decomposition', 'edge cases', 'exception handling', 'extract function', 'function composition', 'graph algorithms', 'hash table usage', 'immutability', 'logical errors', 'loop efficiency', 'map filter reduce', 'memoization', 'null handling', 'off by one', 'pure functions', 'recursion patterns', 'redundant work', 'remove duplication', 'simplify conditionals', 'tree operations', 'variable renaming'

Each problem in the dataset contains a problem id, estimated difficulty level, a natural language problem statement, an initial (sub-optimal) code solution, an ideal solution and unit test cases.

Here is an example from the 'redundant work' category in the training dataset. The problem statement is:

```
You are given a list of tasks, where each task has an ID, a category, and an
initial priority.  You are also given a list of operations.  Each operation is
of the form `('UPDATE_PRIORITY', category_name, new_priority)`, indicating that
all tasks belonging to `category_name` should have their priority changed to
`new_priority`.  If multiple operations target the same category, the latest
operation in the list for that category determines its final priority.  Your
objective is to calculate the total sum of the final priorities of all tasks
```

after considering all operations. To optimize, first determine the definitive priority for each category affected by operations. Then, sum the priorities of all tasks, using the determined category priority if available, or the task's original priority otherwise. This avoids redundantly processing updates for tasks.

Here is the initial code that the LLM must improve

Initial Code: algorithm_optimization_redundant_work_3c471b95

```python
import hashlib

def solution(tasks, operations):
    log_file = open("operation_audit.log", "a")

    updated_priorities_list = []

    for op_type, category_name, new_priority in operations:
        if op_type == 'UPDATE_PRIORITY':
            hashed_category_name = hashlib.md5(
                category_name.encode('utf-8')
            ).hexdigest()

            log_file.write((
                f"Processing UPDATE_PRIORITY for {category_name}"
                f" (hashed: {hashed_category_name})\n"
            ))

            found = False
            for i in range(len(updated_priorities_list)):
                if updated_priorities_list[i][0] == hashed_category_name:
                    updated_priorities_list[i] = (hashed_category_name,
                    ↪  new_priority)
                    found = True
                    break
            if not found:
                updated_priorities_list.append((hashed_category_name, new_priority))

    total_priority_sum = 0
    for task in tasks:
        category = task['category']
        hashed_category = hashlib.md5(category.encode('utf-8')).hexdigest()

        priority_to_add = task['priority']

        for stored_hashed_category, stored_priority in updated_priorities_list:
            if stored_hashed_category == hashed_category:
                priority_to_add = stored_priority
                break

        total_priority_sum += priority_to_add

    return total_priority_sum
```

B More Code Examples

Here are some more side-by-side code examples from Qwen 2.5 3B trained with and without the quality reward component. Note that these may not be functionally identical, with some potentially containing bugs.

Without Quality Reward

```python
import numbers

def solution(price_catalog, shopping_list):
  total_cost = 0.0
  for item in shopping_list:
    item_found_in_catalog = False
    if item in price_catalog:
      item_found_in_catalog = True
      price = price_catalog[item]
      total_cost += price

  return total_cost
```

With Quality Reward

```python
def solution(price_catalog, shopping_list):
    total_cost = 0.0

    for item in shopping_list:
        if item in price_catalog:
            total_cost += price_catalog[item]

    return total_cost
```

Without Quality Reward

```
import hashlib

def solution(data, threshold):
    if not data:
        return "Quicksort"

    data.sort()

    inversion_log = ""
    inversions1 = 0
    for i in range(len(data) // 2):
        for j in range(i + 1, len(data)):
            if data[i] > data[j]:
                inversions1 += 1
                inversion_log += f"Inversion: ({data[i]} > {data[j]}) at indices
                ↪  ({i}, {j}); "

    inversions2 = 0
    for i in range(len(data) // 2, len(data)):
        for j in range(i + 1, len(data)):
            if data[i] > data[j]:
                inversions2 += 1
                inversion_log += f"Inversion: ({data[i]} > {data[j]}) at indices
                ↪  ({i}, {j}); "

    inversions = inversions1 + inversions2

    if inversions <= threshold:
        return "Insertion Sort"
    else:
        return "Quicksort"
```

With Quality Reward

```
import hashlib

def solution(data, threshold):
    n = len(data)

    hasher = hashlib.md5()
    hasher.update(str(data).encode('utf-8'))
    data_hash = hasher.hexdigest()

    total_inversions = 0
    for i in range(n // 2):
        for j in range(i + 1, n):
            if data[i] > data[j]:
                total_inversions += 1

    for i in range(n // 2, n):
        for j in range(i + 1, n):
            if data[i] > data[j]:
                total_inversions += 1

    if total_inversions <= threshold:
        return "Insertion Sort"
    else:
        return "Quicksort"
```

Without Quality Reward

```
def solution(numbers):
    if not numbers:
        return 1

    result = 1
    for num in numbers:
        if num == 0:
            return 0
        elif isinstance(num, int) and abs(num) <= 1000000:
            result *= num
        elif num == 0:
            return 0

    if abs(result) > 1000_000:
        return "Overflow"

    return result
```

With Quality Reward

```
import math
from typing import List

def solution(numbers: List[int]) -> int:
    MAX_PRODUCT = 1_000_000

    if numbers is None or type(numbers) != list:
        return 1

    # Handle empty list
    if not numbers:
        return 1

    result = 1
    for num in numbers:
        if not isinstance(num, int):
            continue

        if num == 0:
            return 0

        result *= num

        # Check for overflow after each multiplication
        if abs(result) > MAX_PRODUCT:
            return "Overflow"

    return result
```