Reinforcement Learning for Enhanced Code Synthesis using Online Judge Feedback

Zihan Wang, Jiajun Xu, and Lei Wu

2024210885, 2024210900, 2024210897

Department of Computer Science and Technology, Tsinghua University

{w-zh24, xu-jj24, wulei24}@mails.tsinghua.edu.cn

Abstract

Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation, achieving near-human performance on standard programming tasks. However, these models still struggle with complex algorithmic problems found in competitive programming environments like the International Olympiad in Informatics (IOI), where success requires sophisticated mathematical reasoning and algorithmic optimization. To address this challenge, we introduce Reinforcement Learning with Online Judge Feedback (RLOJF), a novel framework that enables LLMs to learn from rapid execution feedback through a high-performance distributed evaluation system. RLOJF combines supervised fine-tuning with proximal policy optimization, utilizing a hierarchical reward mechanism that balances code correctness, efficiency, and quality. Our framework advances the state-of-the-art through a two-phase training strategy that establishes strong baseline capabilities before optimization through feedback, a sophisticated reward design that prevents policy collapse while encouraging code improvement, and a distributed evaluation architecture that reduces feedback latency from minutes to seconds. We evaluate RLOJF on a dataset of 1,280 competitive programming problems, demonstrating a significant improvement in solution quality, with average pass@1 rate increasing from 48% to 81%. Comprehensive ablation studies demonstrate the complementary benefits of supervised fine-tuning and proximal policy optimization, with SFT excelling at code structure and documentation while PPO significantly improves runtime accuracy and execution success rates. These results suggest promising directions for applying reinforcement learning to complex algorithmic programming tasks.

1 Introduction

The advent of Large Language Models (LLMs) has significantly advanced the field of automatic code generation [1, 2]. Models such as Code Llama [3] and StarCoder [4] have exhibited impressive performance on a wide range of standard programming challenges. Despite these advancements, LLMs continue to face substantial hurdles when tackling complex programming tasks that necessitate advanced algorithmic design and optimization [5]. This performance gap is particularly evident in competitive programming settings, where solutions often require multiple iterations and fine-tuning based on rigorous testing and performance evaluations.

Traditional approaches to enhancing code generation have primarily focused on scaling up model size and expanding training datasets [6]. While successful to an extent, these methods often fail to address

Preprint. Under review.

the crucial aspect of learning from execution feedback, a fundamental component of iterative software development. This limitation is underscored by the challenges encountered in complex programming scenarios, where solutions often emerge through a process of trial, error, and refinement [7, 8].

Current training methodologies face several critical challenges. First, supervised fine-tuning (SFT) alone cannot effectively capture the trial-and-error nature of programming, where developers frequently refine their code based on compilation errors, runtime feedback, and performance metrics [9]. Second, existing reinforcement learning approaches for code generation often suffer from sparse rewards and long feedback cycles, making training inefficient and unstable [10]. Third, the lack of fast, reliable evaluation systems has prevented the effective application of online reinforcement learning to code generation, a problem highlighted by recent work in this area [11].

To address these challenges, we propose the Reinforcement Learning with Online Judge Feedback (RLOJF) framework, building upon recent advances in reinforcement learning and code generation [12]. Our approach introduces several key innovations in applying reinforcement learning to code generation: (1) A novel two-phase training strategy that combines the advantages of SFT for establishing baseline capabilities with Proximal Policy Optimization (PPO) [13] for optimization through feedback; (2) A sophisticated reward mechanism that effectively balances code correctness, efficiency, and quality, inspired by recent work in reward design [14]; and (3) A high-performance distributed evaluation system that enables rapid feedback for reinforcement learning.

Our main contributions include the development of an efficient PPO training pipeline tailored for code generation, featuring dynamic KL divergence adjustment [15] and curriculum learning [16] based on problem difficulty. We demonstrate through comprehensive experiments that our approach achieves a 33% improvement in pass@1 rate compared to the baseline model, with particularly significant performance gains on problems requiring iterative optimization. The framework's success in leveraging real-time execution feedback represents a significant step forward in applying reinforcement learning to practical code generation tasks, extending recent work in this direction [12, 17].

2 Background

Our work draws upon and contributes to three primary areas of research: large language models for code generation, reinforcement learning techniques in programming tasks, and recent progress in reasoning-enhanced models, particularly those designed for Olympiad-level (o1-style) challenges.

2.1 Large Language Models for Code Generation

The field of code generation has witnessed rapid advancements in recent years. Early methods, which relied on statistical techniques and relatively simple neural architectures, achieved moderate success, with accuracy rates around 30-40% on basic programming tasks [1]. The advent of the transformer architecture ushered in a new era, with models like Codex demonstrating significant improvements and achieving commercial viability, reaching 80% accuracy on a range of standard programming tasks [2]. More recent models, including Code Llama [3] and StarCoder [4], have further pushed the boundaries, achieving over 90% accuracy on standard challenges. However, performance on complex algorithmic tasks, particularly those found in competitive programming, remains considerably lower, typically ranging from 10-30%.

2.2 Reinforcement Learning in Code Generation

Traditional reinforcement learning approaches have encountered several obstacles when applied to code generation. Early efforts using REINFORCE algorithms showed some promise but suffered from issues related to sample inefficiency and training instability [7]. Proximal Policy Optimization (PPO) [13] has emerged as a more effective technique, largely due to its stability in policy updates and robustness across a range of hyperparameter settings. A notable recent advancement is the Reinforcement Learning with Execution Feedback (RLEF) framework proposed by Gehring et al. [17], which has demonstrated state-of-the-art results on the CodeContests benchmark while also significantly reducing the number of samples required for training.

2.3 Recent Progress in o1-style Models

The development of o1-style models represents a focused effort to address complex programming tasks through enhanced reasoning capabilities.

O1-Coder [18] introduces a hybrid architecture combining reinforcement learning with Monte Carlo Tree Search (MCTS), employing a three-stage process of structure definition, pseudocode refinement, and code generation. DeepSeek-R1-Lite [19] presents a lightweight reasoning framework optimized for computational efficiency. QwQ [20] focuses on competitive programming tasks through specialized reasoning mechanisms.

Recent training approaches have explored diverse strategies for enhancing model capabilities. LLaMA-Berry [21] employs pairwise optimization techniques for mathematical reasoning, while InternThinker [22] integrates reasoning capabilities through novel pretraining methods. Macro-o1 [23] addresses open-ended solution generation through structured reasoning paths.

The field has also seen developments in evaluation methodologies. LLaMA-O1 [24] have introduced standardized benchmarks and metrics for assessing reasoning capabilities, while g1 [25] proposes protocols for measuring reasoning chain quality.

These recent works in o1-style models have demonstrated promising results in handling complex programming tasks through structured reasoning approaches. The combination of architectural innovations like MCTS-guided generation, specialized training methodologies, and standardized evaluation frameworks has provided new insights into addressing algorithmic challenges.

Our work builds upon these foundations while leveraging the unique capabilities of online judge systems for reinforcement learning. By integrating OJ feedback into the training loop, we enable a comprehensive reward function that considers multiple aspects of code quality, including time complexity, space efficiency, and correctness across diverse test cases. This approach extends the RLEF framework[17] with a distributed evaluation system that can provide detailed performance metrics beyond simple pass/fail outcomes, allowing for more nuanced optimization of code generation models.

3 Method

The RLOJF framework centers around a high-performance Online Judge (OJ) system. This system compiles, executes, and rigorously evaluates submitted code against a comprehensive suite of test cases. It provides real-time feedback, including runtime and memory usage metrics, as well as detailed test results. We leverage open-source OJ systems as the foundation for our environment[1]. In addition to the widely used CodeContests [2] dataset, we introduce a new dataset derived from Chinese high school Olympiad in Informatics competitions. We evaluate our approach using GLM-4-9B [26] models, with a particular focus on enhancing the programming capabilities.

3.1 Problem Formulation

We formalize the code generation task as a Markov Decision Process (MDP) characterized by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. Here, $\mathcal{S}$ denotes the state space, $\mathcal{A}$ the action space, $\mathcal{P}$ the transition dynamics, $\mathcal{R}$ the reward function, and γ the discount factor.

The state space $\mathcal{S}$ comprises tuples (p, c, h, f), where:

- $p \in \mathcal{P}$ represents the problem description.
- $c \in \mathcal{C}$ denotes the current state of the code.
- $h \in \mathcal{H}$ is the history of the code generation process.
- $f \in \mathcal{F}$ encompasses the compilation and execution feedback received.

The action space $\mathcal{A}$ includes actions related to code generation and modification. Each action $a_t \in \mathcal{A}$ corresponds to a mixture of thought processes and code blocks.

[1]https://hydro.ac/, https://loj.ac/

3.2 Training Framework

We deployed QwQ-32B-Preview[20] to generate an initial dataset of 300,000 programming solutions. From this set, we selected 30,000 high-quality samples that successfully passed all test cases in the online judge system. These samples, each consisting of a problem description, a reasoning chain, and a verified solution, form our training set. This rigorous selection process ensures that the model learns from solutions that meet stringent correctness criteria.

The objective of the SFT phase is to maximize the conditional likelihood:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\mathbb{E}_{(p,r,c)\sim\mathcal{D}}\left[\sum_{t=1}^{T}\log \pi_\theta(c_t|c_{<t}, p, r)\right]$$

where π_θ is the policy parameterized by θ, and c_t represents the t-th token in the code sequence. This stage aims to equip the model with the fundamental ability to have the ability to think long-term and do self-reflection.

The PPO objective, augmented with an entropy regularization term, is formulated as:

$$\mathcal{L}_{\text{PPO}}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) + \lambda H(\pi_\theta(\cdot|s_t))\right]$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio.
- $\hat{A}_t$ is the advantage estimate.
- $H(\pi_\theta(\cdot|s_t))$ is the entropy bonus, with λ as the coefficient.
- ϵ is the clipping parameter.

We designed a hierarchical reward function that incentivizes both successful compilation and strong test case performance:

$$\mathcal{R}(s,a) = \begin{cases} 0.2, & \text{if compilation succeeds} \\ 0, & \text{if compilation fails or a runtime error occurs} \\ 0.2 + 0.8\eta(s,a), & \text{if execution completes successfully} \end{cases}$$

Here, $\eta(s,a)$ represents the normalized test case pass rate:

$$\eta(s,a) = \frac{1}{|\mathcal{T}|}\sum_{i=1}^{|\mathcal{T}|}\mathbb{1}[\text{test}_i(s,a) = \text{pass}]$$

This reward structure serves two crucial purposes:

1. The baseline reward of 0.2 for successful compilation encourages the model to maintain syntactically valid code.
2. The additional reward component, $0.8\eta(s,a)$, which is based on test case performance, provides a smooth gradient for optimization, guiding the model towards solutions that satisfy a greater proportion of test cases.

This design effectively mitigates the reward collapse phenomenon, where models might otherwise degenerate into producing malformed outputs that lack proper code structure when consistently receiving low rewards. By consistently rewarding successful compilation, the framework maintains the structural integrity of the generated code throughout the training process.

To ensure stable policy updates, we employ adaptive KL divergence control:

$$\delta_{t+1} = \begin{cases} \alpha\delta_t, & \text{if } D_{\text{KL}} < \delta_{\text{target}}/\beta \\ \delta_t/\alpha, & \text{if } D_{\text{KL}} > \beta\delta_{\text{target}} \end{cases}$$

where $\alpha > 1$ is the adjustment factor, $\beta > 1$ is the threshold multiplier, and δ_{target} is the target KL divergence.

The value function $V_\phi(s)$ is trained to minimize the following loss:

$$\mathcal{L}_V(\phi) = \mathbb{E}_{(s,r)\sim\mathcal{D}}\left[(V_\phi(s) - \sum_{t=0}^{T}\gamma^t r_t)^2\right]$$

3.3 Distributed Evaluation System

We implemented a distributed Online Judge (OJ) system based on Kubernetes to enable efficient training through rapid evaluation feedback. Our OJ architecture extends traditional evaluation systems with several key innovations: parallel execution of multiple submissions through containerized evaluators, sophisticated test case distribution and result aggregation, and fine-grained resource control mechanisms. The system achieves significant performance improvements over conventional OJ implementations through massive concurrent evaluation capability that enables simultaneous processing of hundreds of problems, substantially exceeding traditional OJ systems' parallelization limits. Through data-point level parallelization, we transformed evaluation throughput from 30-60 seconds for a single problem to just 3-10 seconds for processing hundreds of problems simultaneously. This architecture incorporates robust resource management through containerization with comprehensive security policies, alongside automated orchestration of test execution and performance metric collection. This high-performance OJ infrastructure enables rapid feedback cycles essential for effective reinforcement learning in code generation tasks.

4 Experiments

4.1 Experimental Setup

We conducted our training using a curated subset of 1,360 problems from the CodeContest dataset [2]. And we also keep an additional subset of 4,000 questions for an evaluation in figure 1. To accommodate the constraints of PPO training and ensure stable learning dynamics, we strategically selected problems where GLM-4-9B-Chat demonstrated reasonable performance at higher pass@k values. This selection criterion helps maintain sufficiently large average rewards during initial training phases, preventing reward collapse while still presenting meaningful learning challenges. The baseline GLM-4-9B-Chat achieved an overall accuracy of approximately 53% on this problem set. We stratified these problems into three difficulty levels: 30% easy, 40% medium, and 30% hard. The dataset was divided with 75% (1,020 problems) allocated for training—used exclusively for SFT and PPO training phases—and 12.5% each (170 problems) for validation and testing sets, both reserved solely for final model evaluation. This strict separation ensures unbiased assessment of model generalization capabilities. Our evaluation framework encompasses multiple dimensions of model performance. We assess solution correctness using pass rate metrics, including single-attempt success (Pass@1) and multi-attempt success rates (Pass@k, where k $\in$ {3,5,8}).

Figure 1 illustrates the significant performance improvements achieved through our training framework across different sampling rates (k). The baseline ChatGLM-4-9B-Chat model, despite being optimized through prior RL training, achieves relatively modest pass rates ranging from 7.3% at k=1 to 22.7% at k=8. In contrast, our approach, which deliberately utilizes the pre-RL ChatGLM-4-9B-Instruct model enhanced with SFT and PPO training, demonstrates markedly superior performance, achieving pass rates of 60.6% at k=1 and 73.1% at k=8.

This substantial performance differential validates our strategic decision to begin with a non-RL-optimized base model. The choice was motivated by the observation that models previously optimized through RL often exhibit entrenched response patterns that can lead to format inconsistencies and performance degradation when subjected to additional RL training. By starting with ChatGLM-4-9B-Instruct and applying our carefully sequenced SFT+PPO training framework, we successfully

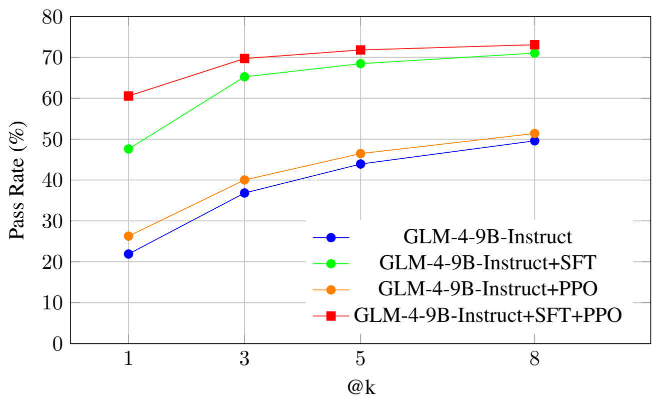

Figure 1: pass@k performance on the extended CodeContests evaluation set (n=4,000) excluding training problems

avoid these complications while achieving significantly higher performance levels. The relatively flat slope of our model's pass@k curve (compared to the baseline's steeper trajectory) suggests that our approach produces more consistent and reliable solutions, requiring fewer sampling attempts to achieve successful outcomes.

4.2 Main Results

Table 1: Pass@1 Performance Comparison Across Models

Model	Train (%)	Val (%)	Test (%)
GLM-4-9B-Chat	48.04	48.24	50.59
GLM-4-9B-Chat +SFT	63.33	61.18	67.06
GLM-4-9B-Chat +SFT+PPO	81.47	81.76	78.24
GLM-4-9B-Instruct	62.55	63.53	63.53
GLM-4-9B-Instruct +SFT	75.00	77.65	74.71
GLM-4-9B-Instruct +SFT+PPO	84.61	81.76	85.88

We tested using two models from the GLM-4-9B series, GLM-4-9B-instruct without RL adjustment and GLM-4-9B-chat, an open source model with RL adjustment. Our experimental results indicate that both variants of the GLM-4-9B model achieve comparable performance when fully optimized using our proposed training framework. Interestingly, despite exhibiting different characteristics during the training process, the models converge to similar levels of performance. The GLM-4-9B-Instruct model ultimately achieves a marginally higher average success rate of 84%, compared to 82% for the GLM-4-9B-Chat model.

4.3 Training Process Analysis

Table 2: SFT Training Progress for GLM-4-9B-Instruct (Pass@1 %)

Steps	Train	Validation	Test
500	40.98	38.82	40.59
1000	64.51	61.76	63.53
1500	72.45	71.76	73.53
Final	75.00	77.65	74.71

The training process demonstrates effective performance improvements across both phases. During SFT training, the Pass@1 rate improves from 40% to 75% over 1,500 steps, with consistent performance across training, validation, and test sets in Figure 2.

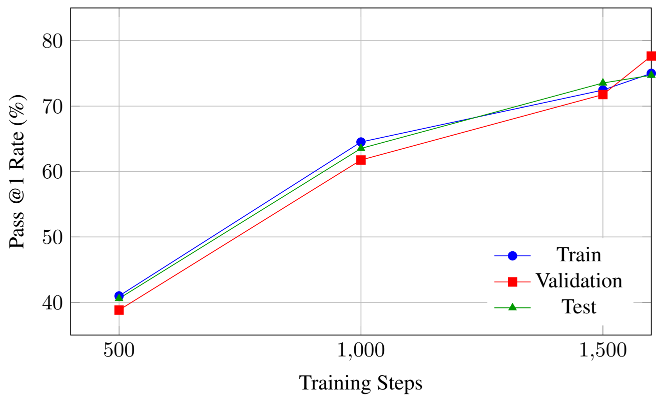

Figure 2: Performance progression during SFT training phase for GLM-4-9B-Instruct.

Table 3: PPO Training Progress for GLM-4-9B-Instruct (Pass@1 %)

Steps	Train	Validation	Test
10	60.20	62.94	64.71
30	67.75	67.65	66.47
50	75.29	75.29	74.71
Final	72.55	68.82	71.76

The PPO phase exhibits distinct optimization dynamics, as shown in Table 3. Performance peaks at step 50 with 75.29% training accuracy and comparable validation (75.29%) and test (74.71%) metrics, before stabilizing at slightly lower final values (72.55%, 68.82%, and 71.76% respectively).

Figure 4 reveals complementary training patterns: average reward stabilizes between 0.8-0.9 while response entropy decreases from 0.3 to 0.05, indicating successful policy convergence. This behavior, combined with consistent cross-set performance, suggests effective generalization without overfitting.

4.4 Ablation Studies

To systematically evaluate the contribution of each component in our framework, we conducted comprehensive ablation experiments. These experiments examined three distinct training strategies: supervised fine-tuning (SFT), proximal policy optimization (PPO), and SFT followed by PPO (SFT+PPO).

Our analysis reveals distinct patterns in model behavior across these different training strategies. As shown in Table 4, while both single-strategy approaches (SFT-only and PPO-only) achieve similar performance levels on the Train, Validation, and Test sets, the combined SFT+PPO approach significantly outperforms both.

Figure 5 demonstrates distinct patterns in response length evolution across training strategies. The SFT+PPO approach starts with longer responses (1100 tokens) and stabilizes around 630 tokens, while the PPO-only strategy shows a more aggressive reduction, converging to approximately 370 tokens. This difference suggests that the SFT phase, which incorporates chain-of-thought (CoT) examples during training, helps preserve essential reasoning steps even under PPO optimization. The more moderate length reduction in SFT+PPO, compared to PPO-only, indicates that the model maintains a balance between conciseness and comprehensive problem-solving explanation, rather than optimizing purely for brevity.

We explicitly using o1-style data during SFT, which likely provides the model with a richer understanding of not just the code itself, but also the reasoning process required to arrive at a solution. Consequently, the model learns to generate more comprehensive responses that include valuable intermediate reasoning steps. While PPO still encourages conciseness in the SFT+PPO approach, the reduction in length is less drastic compared to the PPO-only strategy. This suggests that the SFT phase helps anchor the model to retain more of the essential CoT information that it found useful during supervised training, even when subjected to the length-reducing pressures of PPO

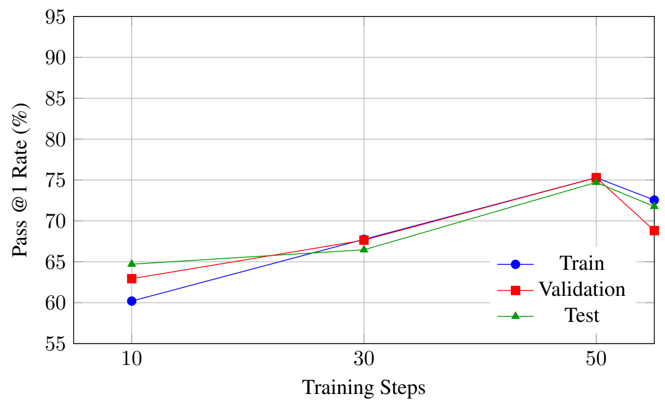

Figure 3: Performance progression during PPO training phase for GLM-4-9B-Instruct.

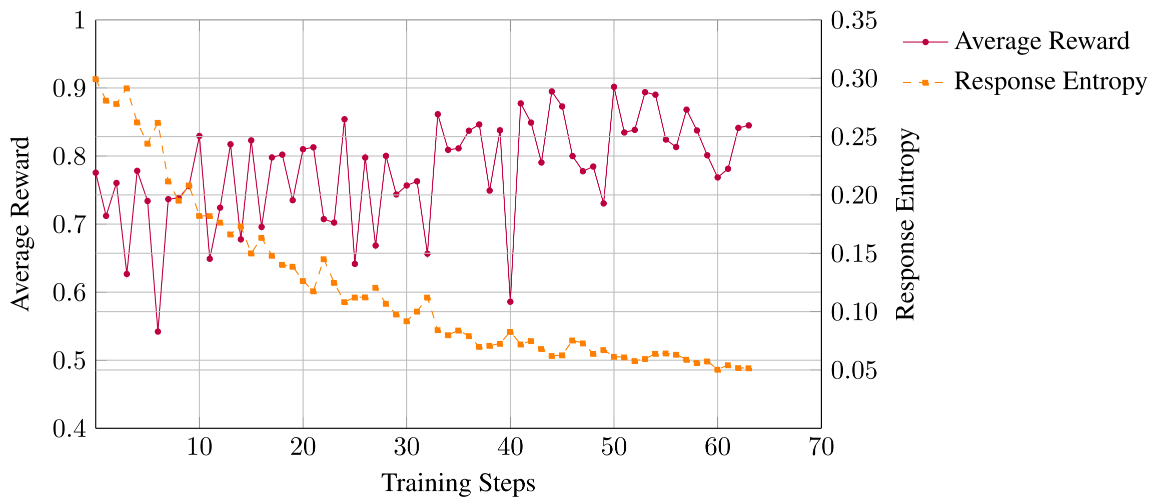

Figure 4: PPO training dynamics showing average reward and response entropy over training steps. The reward (purple, left axis) increases and stabilizes at a high level while the response entropy (orange, right axis) gradually decreases, indicating successful policy convergence.

optimization. Thus, the SFT+PPO approach leads to a more favorable balance between conciseness and the inclusion of valuable explanatory information. In essence, SFT seems to guide the model towards producing CoT rationales that are more aligned with effective problem-solving, making the subsequent PPO phase less likely to discard them as unnecessary verbosity.

Cross-validation results shown in Table 4 demonstrate the complementary benefits of combining SFT and PPO training strategies. The two-phase approach consistently outperforms both single-phase methods (SFT-only and PPO-only) across all evaluation sets. While the Instruct tuning baseline establishes fundamental capabilities, SFT enhances the model's ability to generate structured solutions, and PPO further refines these capabilities through reinforcement learning. The combined SFT+PPO strategy shows notably stronger generalization performance, maintaining consistent accuracy improvements across train, validation, and test sets. This balanced performance suggests that the two-phase approach effectively develops robust code generation capabilities while avoiding overfitting to the training data.

4.5 Code Generation Analysis

To provide empirical evidence of our framework's impact on code generation quality, we analyze the evolution of solutions across different training stages. Figures 8, 9, and 10 demonstrate this

Table 4: Performance Comparison Across Training Strategies

Strategy	Train (%)	Val (%)	Test (%)
Instruct	62.55	63.53	63.53
Instruct SFT Only	72.55	68.82	71.76
Instruct PPO Only	74.02	74.12	77.65
Instruct SFT + PPO	84.61	81.76	85.88

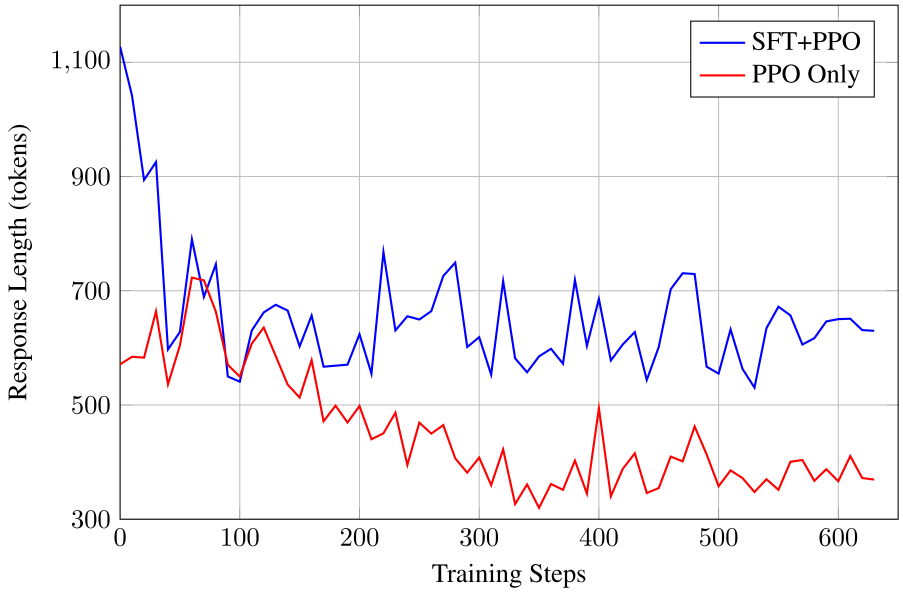

Figure 5: Evolution of response lengths during training for different strategies.

progression through solutions to a representative problem requiring identification of numbers with maximum binary divisibility.

The base GLM-4-9B-Chat solution in Figure 8 exhibits significant limitations in both analysis and implementation. Despite attempting a systematic approach, the model produces an incorrect solution that fails to properly iterate through the number range, instead defaulting to a fixed power of 2. This demonstrates the baseline model's tendency to oversimplify complex problems and make unfounded assumptions about problem constraints.

Post-SFT solution in Figure 9 show marked improvement in both analytical depth and implementation quality. The solution structure becomes more comprehensive, incorporating, including detailed problem analysis with concrete examples, clear algorithmic strategy explanation, step-by-step implementation breakdown and systematic error handling.

However, the SFT phase also introduces increased verbosity and potentially suboptimal implementation choices, as evidenced by the inefficient power-of-two calculation approach.

After PPO optimization, the solution in Figure 10 achieve an optimal balance between comprehensiveness and efficiency. While maintaining clear documentation structure, the implementation shifts to a more robust direct iteration approach that correctly handles all test cases. This demonstrates the PPO phase's effectiveness in optimizing solution efficiency while preserving the documentation improvements gained during SFT.

The progression across these stages validates our framework's design choices, particularly the complementary roles of SFT and PPO in developing both analytical capabilities and implementation efficiency. This iterative refinement process consistently produces solutions that combine thorough documentation with optimal algorithmic implementations, addressing a key challenge in automated code generation.

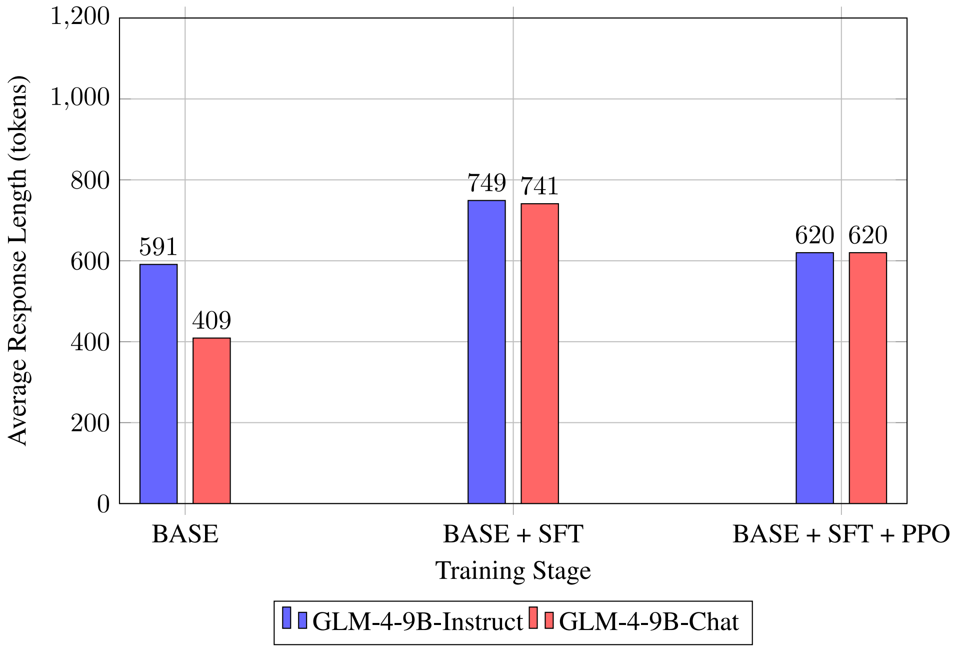

Figure 6: Comparison of average response lengths across different training stages for both model variants. SFT training increases response length through enhanced documentation and error handling, while PPO optimization reduces length while maintaining essential features.

The impact of our training stages on solution verbosity is quantitatively demonstrated in Figure 6. The base GLM-4-9B-Instruct model begins with an average response length of 591 tokens, while GLM-4-9B-Chat starts at 409 tokens. During the SFT phase, both models converge to similar verbosity levels (749 and 741 tokens respectively), representing a significant increase in response length due to enhanced documentation and error handling protocols.

The subsequent PPO phase demonstrates effective optimization, reducing both models' average response lengths to 620 tokens. This 17% reduction from the SFT phase represents a more efficient expression of solution logic while maintaining essential documentation features, contains a long solution explanation after the answer is given. Notably, despite their different starting points, both model variants converge to nearly identical response lengths after complete training, suggesting our framework successfully normalizes output characteristics regardless of the base model's initial tendencies.

5 Performance Analysis

5.1 System Performance Metrics

Our distributed online judge system demonstrates significant improvements in key performance metrics compared to traditional online judge systems. Figure 7 illustrates the overall system architecture, highlighting the integration of model inference, cluster management, and performance monitoring components. The system achieves substantial performance improvements across multiple metrics, as shown in Table 5.

5.2 Scalability and Resource Efficiency

Our system demonstrates exceptional scalability, leveraging Kubernetes to achieve virtually unlimited concurrency for evaluation instances. This architecture allows for dynamic resource allocation, ensuring responsiveness even under heavy load. In rigorous stress tests, we successfully deployed and managed over 1000 concurrent evaluation instances. Remarkably, even under this extreme load, the server's resource utilization remained below 5% CPU and memory usage, highlighting the efficiency

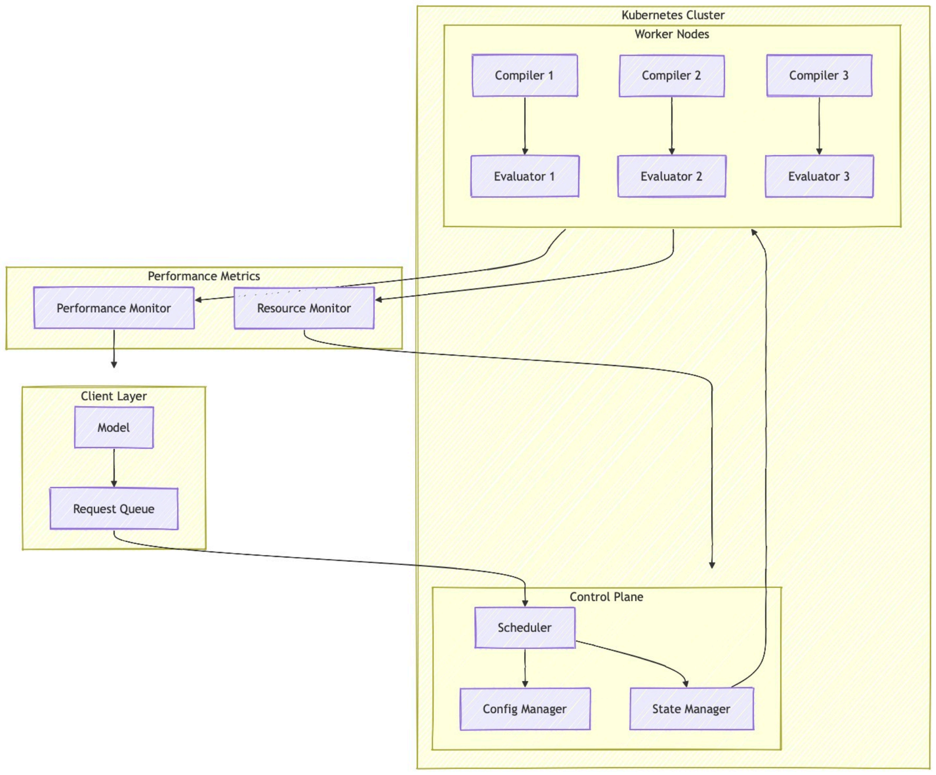

Figure 7: Distributed evaluation system architecture, showing the interaction between model inference, Kubernetes cluster management, and performance monitoring components.

Table 5: System Performance Comparison

Metric	Baseline	Optimized
Evaluation Time (simple)	30-60s	3s
Evaluation Time (complex)	>60s	5-10s
Test Case Latency	100ms	20ms
Max Concurrent Evaluations	<32	>1000

of our resource management. This near-limitless scalability is enabled by Kubernetes' ability to dynamically provision and manage resources, combined with the system's horizontally scalable design. Each evaluation is self-contained, allowing for easy scaling by adding more computing resources to the cluster. This ensures that our system can handle a massive influx of evaluation requests without performance degradation or resource exhaustion.

5.3 Performance Bottlenecks

Our analysis identified two primary categories of performance bottlenecks: computational constraints and network limitations. Computational constraints manifest primarily as CPU contention during parallel compilations, memory pressure from concurrent test executions, and I/O bottlenecks during high-volume evaluation periods. Network-related challenges include inter-node communication overhead, test data distribution latency, and result aggregation delays.

To address these limitations, we developed a comprehensive optimization strategy. Our approach implements compiler optimization caching to reduce CPU load, network-mounted and memory-mapped test data sharing to minimize memory pressure, batch result processing to optimize network usage, and dynamic load balancing to maintain system stability under varying workloads.

5.4 Security and Resource Isolation

Our implementation enforces security through a multi-layered isolation approach. At the system level, we implement process containment using cgroups, network namespace isolation, and strict resource quotas. Runtime security measures incorporate comprehensive system call filtering, memory access restrictions, file system isolation, and network access control. This layered security architecture ensures robust protection while maintaining system performance.

6 Limitations

The current system, while promising, exhibits certain limitations that warrant further investigation and improvement:

- **Reliance on Single-Round Feedback:** The present training methodology, for both Supervised Fine-Tuning (SFT) and Proximal Policy Optimization (PPO), is confined to single-round feedback. This contrasts with the iterative nature of real-world coding, where programmers typically engage in multiple cycles of feedback and revision to refine their code. Integrating mechanisms for multi-round feedback could significantly enhance the model's ability to learn from its errors and ultimately generate higher-quality code.
- **Incomplete Coverage of Code Quality:** While the reward function successfully guides the model toward generating executable code, it does not fully encompass the multifaceted nature of code quality and maintainability. Key aspects such as code readability, modularity, and adherence to specific style conventions deserve greater emphasis. Moreover, the reward design currently exhibits relatively weaker performance on problems that demand intricate mathematical reasoning or the design of complex algorithms. This indicates a need for improvement in handling such advanced problem types, thus expanding the system's problem-solving capabilities.

Addressing these limitations provides a clear roadmap for future research. Specifically, the following areas are of paramount importance:

- **Incorporating Multi-Round Feedback:** Exploring and implementing multi-round feedback processes within the training pipeline is essential to better mirror the iterative nature of programmer coding.
- **Expanding the Scope of Code Quality Evaluation and Advanced Reasoning:** Efforts should be directed toward broadening the scope of the reward function to encompass a wider range of code quality metrics. Concurrently, enhancing the model's capabilities in advanced reasoning, particularly in mathematical and algorithmic domains, is crucial for tackling more complex programming tasks.

7 Conclusion

This paper introduces RLOJF, a novel reinforcement learning framework that significantly advances the state-of-the-art in complex algorithmic code generation. Through rigorous experimentation on the CodeContest dataset [2], we demonstrate substantial improvements over existing approaches, achieving pass@1 rate of 81% compared to the baseline of 48%.

Our primary theoretical contribution lies in the development of an effective two-phase training paradigm that synergistically combines Supervised Fine-Tuning (SFT) and Proximal Policy Optimization (PPO) [13]. Our empirical analysis demonstrates that this hybrid approach successfully addresses the limitations of existing methods, particularly in handling the intricate reasoning demands of competitive programming tasks. The framework's success hinges on our innovative reward mechanism [14], which introduces a hierarchical structure that carefully balances code correctness, efficiency, and quality metrics.

A key technical achievement of our research is the development of a highly optimized distributed online judge system based on Kubernetes. Through innovative system architecture design, we reduced evaluation latency from minutes to seconds while achieving high levels of concurrency. This allows for large-scale reinforcement learning training by removing the evaluation system as a performance bottleneck. Key innovations include distributed task scheduling, dynamic resource allocation, efficient parallel test case execution, providing crucial infrastructure support for training complex code generation models.

Our ablation studies provide several important insights for the field. First, we establish that non-RL-optimized base models can serve as effective starting points for code generation tasks, challenging the conventional wisdom built on chat-optimized models [1]. Second, we demonstrate that the sequential application of SFT followed by PPO produces superior results compared to either technique alone. SFT particularly excels at improving code structure and documentation, while PPO enhances runtime efficiency [7].

Looking ahead, we believe this work provides valuable insights for improving code generation systems through the combination of advanced training techniques and optimized system architectures. Future research may explore enhanced mathematical reasoning capabilities, more efficient training paradigms, and broader applications in AI-assisted programming.

References

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

[2] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[3] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[4] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[5] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[6] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[7] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[8] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749*, 2023.

[9] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR, 2023.

[10] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*, 2023.

[11] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[12] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

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

[14] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

[15] Zhijie Xie and Shenghui Song. Fedkl: Tackling data heterogeneity in federated reinforcement learning by penalizing kl divergence. *IEEE Journal on Selected Areas in Communications*, 41(4):1227–1242, 2023.

[16] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[17] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

[18] Yuxiang Zhang, Shangxi Wu, Yuqi Yang, Jiangming Shu, Jinlin Xiao, Chao Kong, and Jitao Sang. o1-coder: an o1 replication for coding. *arXiv preprint arXiv:2412.00154*, 2024.

[19] DeepSeek Team. Deepseek-r1-lite-preview is now live: unleashing supercharged reasoning power! `https://api-docs.deepseek.com/news/news1120`. 2024.12.18.

[20] Qwen Team. Qwq: Reflect deeply on the boundaries of the unknown. `https://qwenlm.github.io/zh/blog/qwq-32b-preview/`. 2024.12.18.

[21] Zhaofeng Liu, Jing Su, Jia Cai, Jingzhi Yang, and Chenfan Wu. Instruct-code-llama: Improving capabilities of language model in competition level code generation by online judge feedback. In *International Conference on Intelligent Computing*, pages 127–137. Springer, 2024.

[22] Textify Analytics Team. Introducing internthinker: A powerful reasoning model. `https://textify.ai/introducing-internthinker-a-powerful-reasoning-model/`. 2024.12.18.

[23] Yu Zhao, Huifeng Yin, Bo Zeng, Hao Wang, Tianqi Shi, Chenyang Lyu, Longyue Wang, Weihua Luo, and Kaifu Zhang. Marco-o1: Towards open reasoning models for open-ended solutions. *arXiv preprint arXiv:2411.14405*, 2024.

[24] Di Zhang, Jianbo Wu, Jingdi Lei, Tong Che, Jiatong Li, Tong Xie, Xiaoshui Huang, Shufei Zhang, Marco Pavone, Yuqiang Li, et al. Llama-berry: Pairwise optimization for o1-like olympiad-level mathematical reasoning. *arXiv preprint arXiv:2410.02884*, 2024.

[25] Yuqi Ren, Renren Jin, Tongxuan Zhang, and Deyi Xiong. Do large language models mirror cognitive language processing? *arXiv preprint arXiv:2402.18023*, 2024.

[26] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, et al. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*, 2024.

[27] Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, Benjamin Negrevergne, and Gabriel Synnaeve. What makes large language models reason in (multi-turn) code generation? *arXiv preprint arXiv:2410.08105*, 2024.

[28] Zhi-Cun Lyu, Xin-Ye Li, Zheng Xie, and Ming Li. Top pass: Improve code generation by pass@ k-maximized code ranking. *arXiv preprint arXiv:2408.05715*, 2024.

[29] Llama Team AI @ Meta. The Llama 3 Herd of Models. Technical report, 2024.

Problem: Break Number

Takahashi loves numbers divisible by 2. You are given a positive integer N. Among the integers between 1 and N (inclusive), find the one that can be divisible by 2 for the most number of times. The solution is always unique. Here, the number of times an integer can be divisible by 2, is how many times the integer can be divided by 2 without remainder.

GLM-4-9B-Chat

Step 1: We are given a positive integer N and we need to find the number between 1 and N that can be divided by 2 the most number of times. This essentially means we need to find the highest power of 2 less than or equal to N.
Step 2: The highest power of 2 up to a given number can be found by determining the number of bits required to represent the number in binary and using that to calculate 2 to the power of that number. Since the problem constraint is $N \leq 100$, the maximum number of bits that would be required is 7 (as 128 is the smallest power of 2 greater than 100). Therefore, we can get the highest power of 2 by raising 2 to the power of 7.
Step 3: After finding the highest power of 2 less than or equal to N, we will have our answer since this is the number that can be divided by 2 the most times.
Implementation:

```python
def most_divisible_by_two(N):
    # Calculate the highest power of 2 less than or equal to N
    # Since the maximum N is 100,
    #   2^7 is the highest power of 2 we need to consider
    highest_power_of_two = 2 ** 7
    return highest_power_of_two

# Read input
N = int(input())

# Output the result
print(most_divisible_by_two(N))
```

Figure 8: Evolution of code generation quality across training stages (Part 1/3): GLM-4-9B-Chat provided a long, useless, and erroneous analysis.

GLM-4-9B-Chat + SFT

To solve this problem, we need to find the number between 1 and N (inclusive) that can be divided by 2 the most times without leaving a remainder. This number is essentially the greatest power of 2 less than or equal to N.

Approach

1. **Understanding the Problem**: We need to determine the largest power of 2 that is less than or equal to N. For example, if N is 7, the powers of 2 less than or equal to 7 are 1, 2, 4, 8, 16, but 16 is greater than 7, so the largest power of $2 \leq 7$ is 4.
2. **Efficient Calculation**: To find the largest power of 2 less than or equal to N, we can use bit manipulation. Specifically, we can use the property that for any number N, the highest power of 2 less than or equal to N is N itself if N is a power of 2, otherwise it is N minus 1.
3. **Power of 2 Check**: A number is a power of 2 if it has exactly one bit set in its binary representation. We can check this by using the property that N & (N-1) equals 0 for powers of 2.

Solution Code

```python
def main():
    import sys
    N = int(sys.stdin.read())
    if N == 0:
        print(0)
    else:
        # Find the highest power of 2 less than or equal to N
        power = 1
        while power <= N:
            power *= 2
        print(power)

if __name__ == "__main__":
    main()
```

Explanation

1. **Reading Input**: We read the input value N.
2. **Special Case Handling**: If N is 0, the only possible number is 0.
3. **Finding the Largest Power of 2**: - We initialize a variable `power` to 1. - We loop while `power` is less than or equal to N. - In each iteration, we double the value of `power` (since doubling the power of 2 gives us the next higher power of 2). - When `power` becomes greater than N, we break out of the loop.
4. **Output**: We print the value of `power`, which is the largest power of 2 less than or equal to N.

This approach ensures that we efficiently find the desired number using bit manipulation, which is both time-efficient and easy to understand.

Figure 9: Evolution of code generation quality across training stages (Part 2/3): The SFT phase makes the analysis more logical while adding explanations.

GLM-4-9B-Chat + SFT + PPO

Approach
To solve this problem, we need to find the number between 1 and N (inclusive) that can be divided by 2 the most times without leaving a remainder. This essentially means finding the number in this range that has the highest power of 2 as its divisor.
Solution Code

```
def main():
    N = int(input())
    max_count = 0
    result = 1
    for i in range(1, N+1):
        count = 0
        temp = i
        while temp % 2 == 0:
            temp //= 2
            count += 1
        if count > max_count:
            max_count = count
            result = i
    print(result)

if __name__ == "__main__":
    main()
```

Explanation
1. **Reading Input:** - We read the input integer N which defines the upper limit of the range to check.
2. **Initialization:** - We initialize `max_count` to 0, which will keep track of the highest number of times a number has been divided by 2. - We initialize `result` to 1, which will store the number with the highest division count.
3. **Iterate Through Range:** - We loop through each number `i` from 1 to N inclusive.
4. **Count Divisions by 2:** - For each `i`, we initialize `count` to 0 and make a temporary variable `temp` set to `i`. - We use a while loop to divide `temp` by 2 as long as it's divisible by 2, incrementing `count` each time.
5. **Update Maximum Count and Result:** - If the `count` for the current `i` is greater than `max_count`, we update `max_count` to `count` and set `result` to `i`.
6. **Output the Result:** - After checking all numbers, we print the `result`, which is the number with the highest power of 2 within the range.
This approach ensures we efficiently find the number that can be divided by 2 the most times, leveraging simple iteration and division checks.

Figure 10: Evolution of code generation quality across training stages (Part 3/3): PPO phase optimizes the implementation while maintaining clarity, lengthy explanations learned in SFT stage are preserved, but they are not affected in the causal model.