CodePMP: Scalable Preference Model Pretraining for Large Language Model Reasoning

Anonymous ACL submission

Abstract

Large language models (LLMs) have made significant progress in natural language understanding and generation, driven by scalable pretraining and advanced finetuning. However, enhancing reasoning abilities in LLMs, particularly via reinforcement learning from human feedback (RLHF), remains challenging due to the scarcity of high-quality preference data, which is labor-intensive to annotate and crucial for reward model (RM) finetuning. To alleviate this issue, we introduce CodePMP, a scalable preference model pretraining (PMP) pipeline that utilizes a large corpus of synthesized code-preference pairs from publicly available high-quality source code. CodePMP improves RM finetuning efficiency by pretraining preference models on large-scale synthesized code-preference pairs. We evaluate CodePMP on mathematical reasoning tasks (GSM8K, MATH) and logical reasoning tasks (ReClor, LogiQA2.0), consistently showing significant improvements in reasoning performance of LLMs and highlighting the importance of scalable preference model pretraining for efficient reward modeling.

1 Introduction

Large language models (LLMs) have achieved remarkable progress in natural language understanding and generation, driven by advancements in scalable pretraining and finetuning techniques, including supervised finetuning (SFT) (Wang et al., 2022, 2023a) and Reinforcement Learning from Human Feedback (RLHF) (Bai et al., 2022a; Lightman et al., 2023; Bai et al., 2022b; Gulcehre et al., 2023; Schulman et al., 2017; Rafailov et al., 2024). Despite these advances, enhancing LLMs' reasoning capabilities, particularly for complex logical and mathematical tasks, remains a significant challenge (Wang et al., 2023b; Zhang et al., 2024b). While RLHF has proven effective for improving model performance, its efficacy is constrained by the availability of high-quality preference data, which is expensive and labor-intensive to collect (Cobbe et al., 2021; Zheng et al., 2024). This limitation impedes the scalability of reward model (RM) finetuning, which is instrumental in guiding LLMs toward optimal outputs.

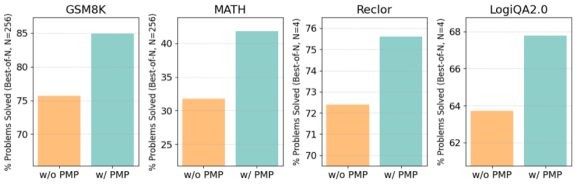


Figure 1: Compared to directly finetuning reward models, CodePMP significantly improves the sample efficiency and capability of reward models, which in turn boosts the generator's(MetaMath-Mistral-7B) reasoning performance (Best-of-N accuracy) across both mathematical reasoning tasks (GSM8K and MATH) and logical reasoning tasks (ReClor and LogiQA2.0).

To alleviate this issue, prior works like Anthropic's Preference Model Pretraining (PMP) (Askell et al., 2021) have proposed improving reward modeling data efficiency by pretraining preference models on large-scale preference data from public sources like Reddit and Wikipedia, followed by an efficient finetuning on limited high-quality human-annotated data. Concurrent work WorldPM (Wang et al., 2025) also explores scaling human preference modeling. However, this approach is less effective for reasoning tasks due to the scarcity of reasoning preference pairs available online. Compared to other tasks, manually annotating preference data for reasoning is inherently more challenging to scale (Zhang et al., 2024b; Zhou et al., 2023), highlighting the urgent need for a scalable PMP approach for reasoning tasks.

In this paper, we propose **CodePMP**, a scalable preference model pretraining pipeline that enhances LLM reasoning abilities using synthesized preference pairs derived from high-quality, publicly available source code. Code, with its inherently

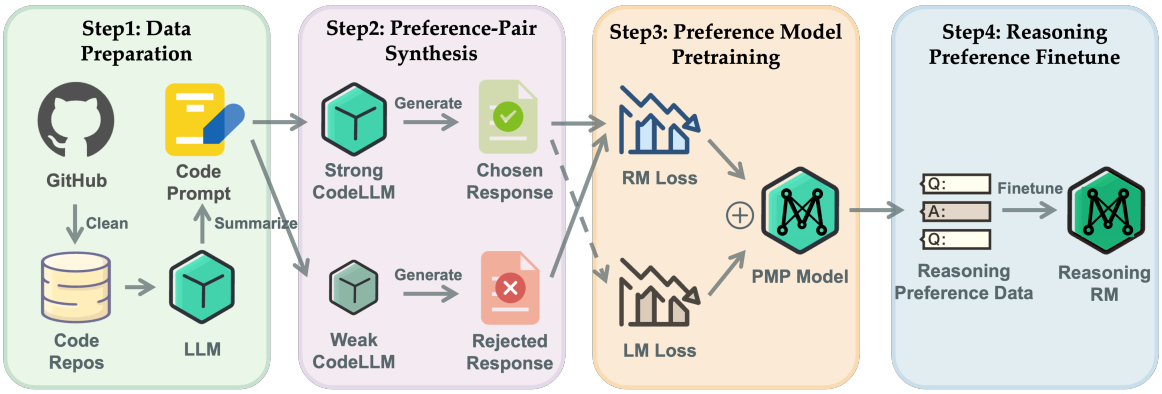


Figure 2: **Overview of CodePMP.** First, raw code collected from GitHub is cleaned and summarized into code prompts (descriptions). Then, a weak CodeLLM generates *rejected* responses while a stronger CodeLLM produces *chosen* responses. Finally, these millions of $\langle chosen, rejected \rangle$ pairs form the preference model pretraining dataset, enhancing both sample efficiency and performance for downstream reasoning tasks.

logical and structured nature, provides rich data suitable for reasoning tasks. Recent works (Zhang et al., 2024b; Aryabumi et al., 2024) also show a strong correlation between code training and reasoning improvements in LLMs. By leveraging the huge amount and diverse coverage of source code available on platforms like GitHub, CodePMP offers a scalable solution for pretraining preference models, thereby improving RM finetuning efficiency and enhancing LLMs' reasoning performance.

Specifically, CodePMP generates preference pairs by synthesizing *chosen* and *rejected* code responses for a given code-related prompt or description using CodeLLMs. A strong CodeLLM produces higher-quality (*chosen*) responses, while a weaker model generates suboptimal or even low-quality (*rejected*) responses. These $\langle chosen, rejected \rangle$ pairs, accumulated in the millions, form a large-scale synthesized preference dataset. This dataset is then used to pretrain the preference model with pairwise ranking objectives (Cobbe et al., 2021; Charniak and Johnson, 2005), providing a good initialization for further finetuning the reward models.

We evaluate CodePMP on widely studied reasoning tasks, including mathematical reasoning tasks such as GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021), as well as logical reasoning tasks like ReClor (Yu et al., 2020) and LogiQA2.0 (Liu et al., 2023). Our experiments show that CodePMP significantly improves RM finetuning accuracy and Best-of-N performance in reasoning tasks, outperforming direct RM finetuning, as highlighted in Figure 1. Moreover, additional results reveal that RMs initialized with CodePMP exhibit greater robustness across different tasks. These results indicate that code-derived preference data provides a scalable, cost-effective solution for enhancing LLM reasoning capabilities while reducing reliance on extensive preference annotation, achieving more effective reward modeling for reasoning tasks.

In summary, our main contributions are:

1. We introduce CodePMP, a scalable method that uses code-derived preference pairs to pretrain preference models, improving sample efficiency and robustness for downstream RM finetuning.
2. We validate that CodePMP significantly improves performance on reasoning tasks, demonstrating that a scalable PMP process positively impacts LLM reasoning abilities.
3. We provide a detailed analysis of key design elements in CodePMP, offering valuable insights for future research in related areas.

2 Preliminaries

Language Modeling Language modeling represents a fundamental task in natural language processing aimed at modeling sequential language data. This is typically implemented through Causal Language Models (Causal LM), which maximize the likelihood of predicting the next token w_t given preceding tokens $w_1, w_2, \ldots, w_{t-1}$. The training process minimizes the negative log-likelihood:

$$\mathcal{L}_{\text{LM}} = -\sum_{t=1}^{T} \log P(w_t | w_1, w_2, \ldots, w_{t-1}) \quad (1)$$

This loss function $\mathcal{L}_{\text{LM}}$ encourages the model to capture underlying patterns in the data. Transformer architectures (Vaswani, 2017) are the standard for Causal LM due to their ability to handle long-range dependencies effectively.

Reward Modeling Reward modeling (RM) is integral to reinforcement learning from human feedback (RLHF), providing scalar reward signals that guide learning based on output quality. The reward model R_θ predicts the quality of an output y given a context x as $s = R_\theta(x, y)$. In preference modeling, RMs predict relative quality by comparing output pairs. A standard approach employs the Pairwise Ranking Loss, which assigns higher scores to preferred (chosen) outputs:

$$\mathcal{L}_{\text{RM}} = -\log\left(\sigma(s_{\text{chosen}} - s_{\text{rejected}})\right) \quad (2)$$

, where $s_{\text{chosen}} = R_\theta(x, y_{\text{chosen}})$ and $s_{\text{rejected}} = R_\theta(x, y_{\text{rejected}})$, and $\sigma(\cdot)$ is the sigmoid function.

Best-of-N Sampling Best-of-N (BoN) sampling enhances LLM reasoning (Cobbe et al., 2021; Lightman et al., 2023) by generating N candidate solutions $\{y_1, y_2, \ldots, y_N\}$ for a given problem, then using a reward model to score and select the highest-scoring candidate:

$$\hat{y} = \arg\max_{y_i \in \{y_1, y_2, \ldots, y_N\}} R_\theta(x, y_i) \quad (3)$$

, where $R_\theta(x, y_i)$ represents the reward score for each candidate y_i. This technique is especially effective in tasks like mathematical problem-solving and logical inference, where selecting the most plausible solution from a diverse set of outputs improves overall accuracy (Wang et al., 2022).

3 Code Preference Model Pretraining

3.1 Model Design

Code Preference Model Pretraining (CodePMP) enhances the sample efficiency of reward models, particularly for reasoning tasks where high-quality preference data is scarce. Traditionally, reward models are finetuned on small, curated datasets, limiting their effectiveness in complex tasks like mathematical reasoning or logical deduction. CodePMP mitigates this limitation by introducing a pretraining phase between basic language model pretraining and finetuning on domain-specific reasoning datasets. This phase leverages a large, diverse dataset of code-preference pairs, enabling the model to learn generalizable patterns and ranking strategies.

CodePMP training involves two components: Reward Modeling (RM) and Language Modeling (LM). In RM, the model is trained on code-preference pairs, learning to assign higher scores to the *chosen* code through a pairwise ranking loss. In LM, only the *chosen* code is used for autoregressive training to maintain the model's general capabilities. The overall loss combines the RM and LM losses, ensuring the model enhances its ranking ability without sacrificing general language modeling performance: $\mathcal{L}_{\text{PMP}} = \mathcal{L}_{\text{RM}} + \mathcal{L}_{\text{LM}}$.

PMP	MathShepherd -pair	Reclor -pair	LogiQA2.0 -pair
	Qwen2-1.5B		
✗	0.7226	0.758	0.7538
✓	**0.8186**	**0.794**	**0.7774**
	Qwen2-7B		
✗	0.8777	0.862	0.8263
✓	**0.9274**	**0.874**	**0.8441**

Table 1: Reward model accuracy comparison: CodePMP-initialized models perform better on reasoning test sets, showing better discrimination ability.

3.2 Data Construction

To enable scalable preference model pretraining, we construct a dataset sourced from GitHub, containing over 1.3 billion code files from GitHub repositories. The CodePMP dataset is constructed through a systematic process. First, raw source code is processed by a description summarizer—typically an instruction-tuned CodeLLM—to generate prompts describing the code's functionality. Two CodeLLMs with different capabilities then generate code snippets based on these prompts:

- **Chosen response**: Generated by a more advanced CodeLLM (e.g., 6.7B parameters).
- **Rejected response**: Generated by a less capable CodeLLM (e.g., 1.3B parameters).

This process yields pairs of code responses—one chosen and one rejected—which are used for preference modeling. This scalable approach significantly enhances pretraining efficiency, improving performance on downstream tasks. The steps of the CodePMP methodology are outlined systematically in Figure 2.

4 Experiments

In this section, we outline the experimental setup and then the experimental results, highlighting that CodePMP is a highly scalable method.

4.1 Experimental Settings

4.1.1 CodePMP Settings

Data Construction We generate code preference pairs by using the deepseek-coder-6.7b-instruct model as the strong CodeLLM to generate *chosen* responses and the deepseek-coder-1.3b-instruct model as the weak CodeLLM to generate *rejected* responses. The constructed CodePMP dataset includes 28 million files and 19 billion tokens. The diverse datasets provide sufficiently broad prompt coverage for preference model pretraining, which is conducive to the generalization of preference models in reasoning tasks. In addition, the average lengths of the *chosen* and *rejected* responses are similar, ensuring that response length does not bias the CodePMP learning process. Details are provided in Appendix.

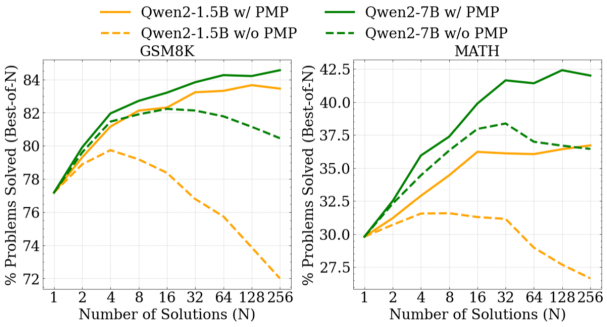
CodePMP Training By default, we initialize the preference models with the publicly available Qwen models (Yang et al., 2024), using different model sizes, specifically Qwen2-1.5B and Qwen2-7B. Detailed hyperparameters for CodePMP training are provided in Appendix.

4.1.2 Reasoning Finetuning Settings

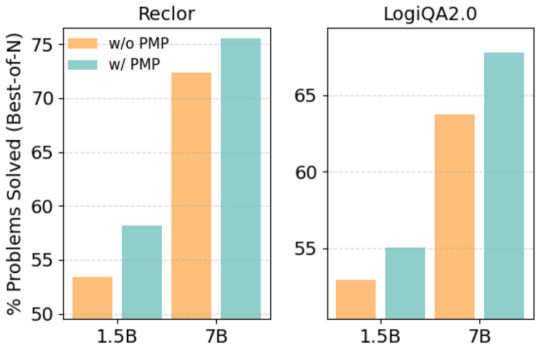
We evaluate CodePMP on mathematical and logical reasoning tasks using dedicated preference datasets. For mathematical reasoning, we finetune reward models on MathShepherd-pair dataset, derived from MathShepherd (Wang et al., 2023b), while logical reasoning models use ReClor-pair and LogiQA2.0-pair datasets, derived from ReClor (Yu et al., 2020) and LogiQA2.0 (Liu et al., 2023) respectively. Each model is finetuned on its corresponding training set and evaluated on its respective holdout test set for accuracy assessment. Implementation details for dataset construction and hyperparameters are provided in Appendix.

4.1.3 Evaluation Settings

Following (Zhang et al., 2024a), we evaluate using two metrics: (1) **RM Accuracy** measures the reward model's ability to distinguish chosen from rejected solutions on holdout test sets, providing insight into the model's ability to classify individual sequences; and (2) **Best-of-N (BoN) Accuracy** assesses the percentage of correct solutions selected by the RM from N candidate responses, evaluating the model's group-wise ranking performance and ability to identify the best answer from multiple candidates. We use MetaMath-Mistral-7B (Yu et al., 2023) as the generator for BoN evaluation.



(a) BoN accuracies on mathematical reasoning.



(b) BoN *(N=4)* accuracies on logical reasoning.

Figure 3: Best-of-N accuracy comparison: CodePMP-initialized models outperform baselines across various N values, showing superior ranking capabilities.

We evaluate on GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021) for mathematical reasoning, and ReClor (Yu et al., 2020) and LogiQA2.0 (Liu et al., 2023) for logical reasoning. For logical reasoning tasks, we use multiple-choice accuracy (equivalent to Best-of-4) where the RM ranks four manually annotated options, as logical reasoning questions typically consist of paragraphs followed by statements to be judged, making standard BoN evaluation challenging.

4.2 Experimental Results

4.2.1 RM Accuracy Results

We first compare RM accuracy on the holdout test set with and without CodePMP initialization. As shown in Table 1, RM finetuned with CodePMP initialization achieves higher accuracy on both 1.5B and 7B models across mathematical and logical reasoning tasks, demonstrating that CodePMP enhances the model's ability to differentiate correct from incorrect reasoning. Moreover, CodePMP exhibits strong generalization, yielding significant improvements across different reasoning tasks.

4.2.2 BoN Accuracy Results

Evaluations across reasoning tasks demonstrate that CodePMP-initialized RMs consistently achieve higher BoN accuracy on both mathematical and logical reasoning tasks for all model sizes (Figure 3). CodePMP models maintain performance advantages even as N increases to 256, while non-CodePMP models exhibit significant accuracy degradation at higher N values, highlighting CodePMP's stability.

This aligns with research on BoN sampling (Chow et al., 2024) that identifies an inflection point where performance typically deteriorates beyond certain N thresholds due to increased base policy stochasticity and verifier misalignment. CodePMP-initialized models demonstrate greater stability at higher N values, suggesting improved alignment with true reward signals and enhanced robustness to noise amplification inherent in large-N sampling.

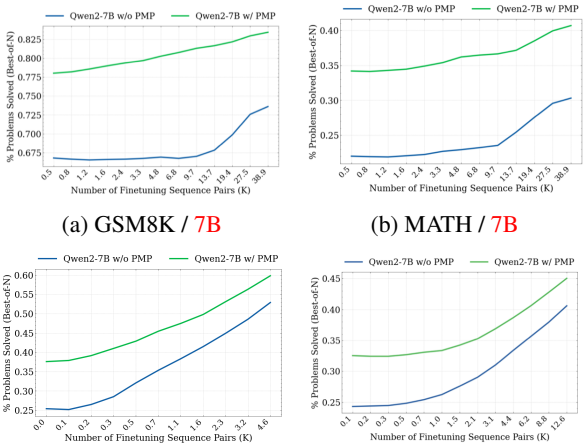
For logical reasoning, the performance gap appears smaller as testing was limited to N=4, while mathematical reasoning extended to N=256, suggesting potential for amplified advantages in logical reasoning with increased N values.

4.2.3 Sample Efficiency Analysis

To assess CodePMP's impact on sample efficiency, we evaluated models with varying finetuning dataset sizes following best practices (Kaplan et al., 2020). Figure 4 shows that CodePMP-initialized models consistently outperform baselines across all dataset sizes, with CodePMP achieving with just 0.5k samples what baseline models require 40k samples to match—an 80× efficiency improvement. This advantage, while diminishing with larger datasets, significantly reduces annotation costs for developing effective reward models.

4.2.4 Scalability Analysis

A key benefit of using code data for PMP is the vast availability of publicly accessible, high-quality code-preference pairs, ensuring diversity. To validate scalability, we vary the number of training pairs for CodePMP and retrain models with different amounts of data. As shown in Figure 5, increasing the number of code-preference pairs consistently improves BoN accuracy in both mathematical and logical reasoning tasks across model sizes, with no sign of diminishing returns. This indicates that further scaling the code-preference data would likely yield additional performance gains, underscoring the importance of building a scalable PMP pipeline.



(a) GSM8K / 7B (b) MATH / 7B

(c) Reclor / 7B (d) LogiQA2.0 / 7B

Figure 4: Sample efficiency comparison for 7B models: CodePMP-initialized reward models achieve higher Best-of-N accuracy with the equivalent sample sizes, showing better data efficiency. Horizontal axis scales by $\sqrt{2}$. Green: with CodePMP; Blue: without CodePMP.

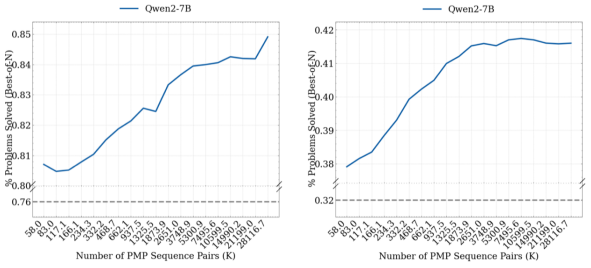
5 Ablation Studies

This section presents a detailed analysis of CodePMP design. Unless otherwise stated, all experiments use the 1B model due to resource limitations and present the results of mathematical reasoning due to page limitation. More ablation studies refer to Appendix.

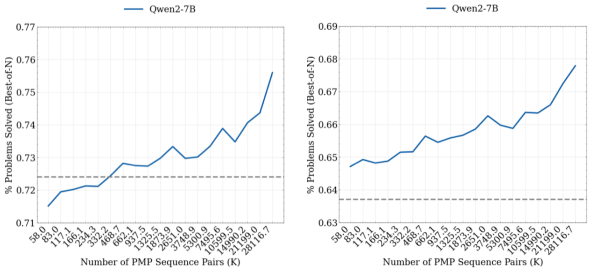
5.1 Impact of Pair Construction

GitHub-Sourced Pairs vs Web-Crawled We compare GitHub-sourced code with web-crawled data (Askell et al., 2021) from platforms such as StackExchange and Reddit. As shown in Figure 6b, GitHub-sourced pairs ("Source Code") consistently outperform those from web platforms ("Webpage"), particularly as the number of solutions (N) increases. Moreover, the performance improvement of GitHub-sourced pairs shows no sign of plateauing, highlighting the importance of diverse, high-quality source code in building a scalable PMP pipeline.

Model Generated Data vs Human Data We compare various pair construction methods generated by different models. In Figure 6a, the samples before the "&" are positive, and those after are negative. "Source Code" refers to the original code snippet, while "1.3B-Des-Clip" indicates that 10% of the code description is removed before being input into a 1.3B CodeLLM to generate a rejected response. The green lines represent CodePMP's



(a) GSM8K / 7B (b) MATH / 7B



(c) Reclor / 7B (d) LogiQA2.0 / 7B

Figure 5: Scaling analysis of CodePMP for 7B models: more code-preference pairs consistently improve Best-of-N accuracy across reasoning tasks without diminishing returns. Horizontal axis scales by $\sqrt{2}$; gray dashed lines show baseline performance without CodePMP.

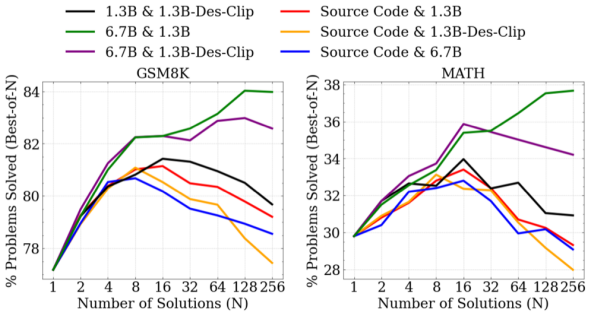
choice. Results show that pairing positive samples from the 7B model with negative samples from the 1.5B model consistently delivers the best performance across all test sets. Given that code execution can generate reliable outputs, future work will explore incorporating execution feedback to create more accurate preference pairs.

5.2 Impact of Loss Function

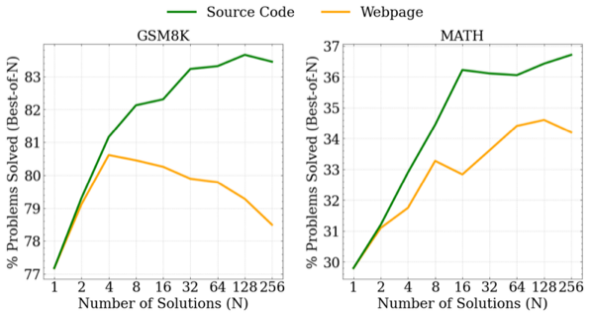
CodePMP integrates both Reward Modeling (RM) and Language Modeling (LM) loss components. To evaluate their contributions, we conducted experiments comparing three configurations: RM loss only, LM loss only, and the combined approach. As shown in Table 2, the combined loss function consistently outperforms single-loss variants across all Best-of-N evaluation settings, with particularly notable improvements on the challenging MATH dataset. This empirical evidence indicates a complementary relationship where RM loss enhances preference ranking while LM loss preserves general language capabilities, collectively yielding more robust reward model performance.

5.3 Cross-Architecture Generalization

To assess CodePMP's generalization capabilities beyond the Qwen architecture family, we evaluated its effectiveness with Gemma2 and Llama3.2 as PMP/RM backbones on GSM8K, MATH, Reclor, and LogiQA-v2 benchmarks. As shown in



(a) Different construction methods.



(b) Different pair sources.

Figure 6: Comparison of BoN accuracy across construction methods and data sources, demonstrating benefits of model-based construction and GitHub code.

BoN	RM Loss	LM Loss	RM + LM Loss
GSM8K			
N=32	0.834	0.8317	**0.8393**
N=64	0.8362	0.8271	**0.8453**
N=128	0.8332	0.8309	**0.8362**
N=256	0.8271	0.8226	**0.8484**
MATH			
N=32	0.344	0.376	**0.418**
N=64	0.358	0.376	**0.424**
N=128	0.366	0.354	**0.434**
N=256	0.362	0.372	**0.41**

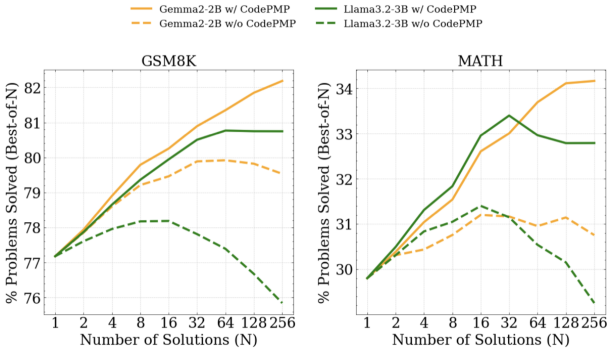
Table 2: Loss function comparison.

Figure 7, CodePMP: (1) Consistently enhances reasoning performance across all model families, and (2) Improves robustness at larger N values, mitigating performance degradation observed in non-initialized models.These results demonstrate that CodePMP generalizes effectively across diverse model architectures, suggesting broad applicability of the approach.

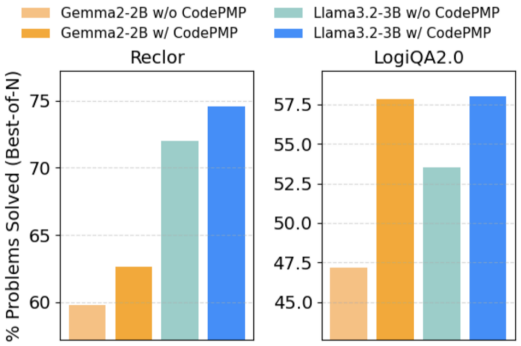
5.4 Performance on Larger Backbone Model

To investigate CodePMP's performance on larger model scales, we applied the technique to Qwen2-72B. Table 3 presents results across mathematical and logical reasoning tasks.

Results show consistent improvements with



(a) BoN accuracies on mathematical reasoning.



(b) BoN *(N=4)* accuracies on logical reasoning.

Figure 7: Cross-architecture performance comparison: CodePMP enhances reasoning performance across different model families (Gemma2 and Llama3.2), showing broad applicability.

BoN	w/o PMP GSM8K	w/ PMP GSM8K	w/o PMP MATH	w/ PMP MATH
N=1	0.7718	0.7718	0.298	0.298
N=4	0.8453	0.8453	0.424	0.424
N=32	0.8529	**0.8628**	0.488	**0.500**
N=256	0.8249	**0.8400**	0.506	**0.514**
BoN	**Reclor**		**LogiQA-v2**	
N=4	0.894	**0.918**	0.7117	**0.7927**

Table 3: Performance comparison on reasoning tasks for Qwen2-72B with and without CodePMP initialization. Note that only $N = 4$ was tested for Reclor and LogiQA-v2.

CodePMP initialization across all benchmarks. Notably, performance gains increase with larger N values on challenging tasks like MATH, indicating that CodePMP's benefits scale effectively to larger model architectures. The significant improvement on logical reasoning tasks further demonstrates CodePMP's scalability and broad applicability.

5.5 Performance on More Powerful Generator

To determine whether CodePMP maintains its effectiveness with more sophisticated generators, we conducted experiments with two advanced models: Qwen2-Math-7B-Instruct (specialized for mathematical reasoning) and Qwen2.5-32B-Instruct (a substantially larger general-purpose model).

BoN	GSM8K w/o PMP	GSM8K w/ PMP	MATH w/o PMP	MATH w/ PMP
N=4	0.8544	**0.8931**	0.690	**0.724**
N=32	0.8446	**0.8795**	0.643	**0.698**
N=256	0.8256	**0.8590**	0.614	**0.690**

Table 4: BoN accuracy with specialized mathematical generator (Qwen2-Math-7B-Instruct).

BoN	GSM8K w/o PMP	GSM8K w/ PMP	MATH w/o PMP	MATH w/ PMP
N=4	0.9604	**0.9688**	0.798	**0.820**
N=32	0.9573	**0.9581**	0.768	**0.792**
N=256	0.9566	**0.9634**	0.752	**0.798**

Table 5: BoN accuracy with large-scale generator (Qwen2.5-32B-Instruct).

Tables 4 and 5 demonstrate that CodePMP's benefits persist across different generator architectures. With the specialized Qwen2-Math-7B-Instruct (Table 4), we observe substantial improvements on both GSM8K and MATH. These gains remain consistent with the much larger Qwen2.5-32B-Instruct model (Table 5), despite it being significantly larger than both the preference pair generation models (7B parameters) and the reward model itself (Qwen2-7B).

These findings demonstrate that reward models trained on synthetic preference data from smaller models can effectively guide more powerful and specialized generators, confirming CodePMP's robustness and cross-scale applicability. This is particularly significant as it suggests that relatively modest investments in reward model training can yield benefits even when deployed with state-of-the-art generation systems.

5.6 Performance on General RM Benchmarks

We further evaluate CodePMP on general reward modeling benchmarks (RMBench) to assess its applicability beyond reasoning tasks. RMBench provides an out-of-domain assessment covering various tasks including summarization, chat quality, and safety. As shown in Table 6, models fine-tuned with PMP consistently outperform those without PMP across various model sizes and tasks.

These results demonstrate that CodePMP enhances performance not only in reasoning and coding tasks but also improves generalization across a broad range of RM benchmarks. These findings provide compelling evidence for CodePMP's broad applicability across multiple domains beyond the

Model	PMP	RMBench Summary	Chat	Chat Hard	Safety	Reasoning
1.5B	✗	0.4154	0.4804	**0.5351**	0.3665	0.2751
	✓	**0.6126**	**0.9050**	0.4364	**0.3698**	**0.6041**
7B	✗	0.5839	0.4972	0.5022	**0.5240**	0.6804
	✓	**0.7668**	**0.9413**	**0.5373**	0.4906	**0.9116**

Table 6: Performance on RMBench shows that CodePMP generalizes well across various general LLM tasks.

reasoning tasks that were our primary focus.

6 Related Works

Reward Modeling In the context of RLHF, reward models (RMs) have traditionally employed ranking models like Bradley-Terry and Plackett-Luce to represent human preferences (Bradley and Terry, 1952; Plackett, 1975; Cobbe et al., 2021; Saunders et al., 2022; Lightman et al., 2023; Wang et al., 2023b; Uesato et al., 2022; Luo et al., 2024; Yu et al., 2024; Stiennon et al., 2020; Nakano et al., 2021). More recently, probability-based approaches have emerged, offering more precise predictions. Additionally, models such as Critique-out-Loud (Ankner et al., 2024) enhance RMs by integrating natural language feedback. Generative reward models (GRMs) further boost sample efficiency. Preference Modeling Pretraining (PMP) (Askell et al., 2021) introduces a novel pretraining phase, utilizing large-scale pairwise ranking data to enhance RM performance. Despite these advancements, many methods are hindered by the reliance on expensive manual annotations or limited datasets, constraining scalability. CodePMP mitigates this by automating preference data generation from code, significantly improving RM sample efficiency and reducing dependency on manual data collection.

Code Training The inclusion of code in LLM pretraining has led to marked improvements in tasks such as commonsense reasoning (Madaan et al., 2022) and mathematical problem-solving (Liang et al., 2022; Shao et al., 2024; Yang et al., 2024). Furthermore, code enhances general reasoning capabilities (Muennighoff et al., 2023; Fu et al., 2022; Ma et al., 2023). Recent studies (Dong et al., 2023; Ma et al., 2023) indicate that incorporating code during supervised finetuning strengthens LLMs, particularly in complex decision-making tasks. CodePMP takes a pioneering approach by utilizing scalable, synthetically generated code preference pairs, reducing the dependence on manual annotation (Dubey et al., 2024; Gemini-Team et al., 2024; Groeneveld et al., 2024; Bi et al., 2024). This methodology enhances sample efficiency and scalability in reasoning-intensive tasks, presenting new opportunities for further improving LLM performance.

LLM Reasoning Improving reasoning capabilities in LLMs remains a significant challenge, with various advanced methods being proposed. Chain of Thought (CoT) prompting (Wei et al., 2022; Fu et al., 2023) improves reasoning by generating intermediate steps, while CoT combined with supervised finetuning (SFT) further enhances performance (Cobbe et al., 2021; Liu et al., 2024; Yu et al., 2023). Other approaches focus on expanding inference time computation, such as problem decomposition (Zhou et al., 2022), search-based methods like MCTS (Xu, 2023), and using LLMs as verifiers (Huang et al., 2022; Luo et al., 2023). Reward models, including outcome-based (ORM) and process-based (PRM), have also shown success, with PRM delivering superior results (Lightman et al., 2023; Wang et al., 2023b). Encouragingly, CodePMP introduces a scalable preference model pretraining phase that can integrate seamlessly with all the aforementioned techniques.

7 Conclusion and Future Works

We propose **CodePMP**, a scalable preference model pretraining method that leverages synthetic code-preference pairs to boost reasoning in large language models. Experiments demonstrate that CodePMP markedly enhances both sample efficiency and performance across diverse reasoning tasks, validating the effectiveness of code-based preference pretraining. Future directions include CodePrMP, which will utilize compiler/interpreter verifiability for low-cost process supervision, and GenPMP, aimed at improving generative reward models through code-based pretraining.

Limitations

Our current implementation has several limitations. First, the synthetic preference pairs rely on models with predetermined parameter sizes, potentially missing nuanced preference signals that more sophisticated approaches might capture. While we demonstrate broad applicability across model families, architectural differences may affect performance in ways not fully explored in this work. Our reliance on GitHub data introduces potential biases stemming from the composition of public repositories. Additionally, our evaluation focuses primarily on mathematical and logical reasoning, leaving the method's effectiveness for other reasoning modalities (e.g., commonsense or causal reasoning) less thoroughly examined. Future work should address these limitations to further enhance the generalizability and robustness of the approach.

Ethics Statement

CodePMP introduces several important ethical considerations. By enhancing LLMs' reasoning capabilities, it could significantly impact decision-making systems that affect human lives, necessitating careful deployment and monitoring. While we utilize publicly available code, we recognize the importance of intellectual property rights and have focused on data with permissive licenses. Our approach reduces reliance on human annotation, potentially mitigating certain biases while possibly introducing others derived from the training data or model preferences. These trade-offs require ongoing evaluation and refinement to ensure fair and beneficial applications. As with any technology that enhances AI capabilities, responsible deployment with appropriate safeguards is essential.

Acknowledgments

We would like to thank all the anonymous reviewers for their insightful comments. We also thank our colleagues for valuable discussions and feedback throughout this research. This work was partially supported by research grants from various organizations, and we are grateful for their support.

References

Zachary Ankner and 1 others. 2024. Critique-out-loud reward models. *arXiv preprint arXiv:2408.11791*.

Viraat Aryabumi, Yixuan Su, Raymond Ma, Adrien Morisot, Ivan Zhang, Acyr Locatelli, Marzieh Fadaee, Ahmet "Ust"un, and Sara Hooker. 2024. To code, or not to code? exploring impact of code in pre-training. *arXiv preprint arXiv:2408.10914*.

Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, and 1 others. 2021. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*.

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, and 1 others. 2022a. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, and 1 others. 2022b. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, and 1 others. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Ralph Allan Bradley and Milton E Terry. 1952. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 173–180.

Yinlam Chow, Guy Tennenholtz, Izzeddin Gur, Vincent Zhuang, Bo Dai, Sridhar Thiagarajan, Craig Boutilier, Rishabh Agarwal, Aviral Kumar, and Aleksandra Faust. 2024. Inference-aware fine-tuning for best-of-n sampling in large language models. *Preprint*, arXiv:2412.15287.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems, 2021. *URL https://arxiv.org/abs/2110.14168*.

Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2023. How abilities in large language models are affected by supervised fine-tuning data composition. *arXiv preprint arXiv:2310.05492*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Yao Fu, Hao Peng, and Tushar Khot. 2022. How does gpt obtain its ability? tracing emergent abilities of language models to their sources.

Yao Fu, Hao Peng, Ashish Sabharwal, Peter Clark, and Tushar Khot. 2023. Complexity-based prompting for multi-step reasoning. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*.

Gemini-Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, and 1 others. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, and 1 others. 2024. Olmo: Accelerating the science of language models. *arXiv preprint arXiv:2402.00838*.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, and 1 others. 2023. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*.

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. *arXiv preprint arXiv:2103.03874*.

Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, and 1 others. 2024. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*.

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, and 1 others. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Hanmeng Liu, Jian Liu, Leyang Cui, Zhiyang Teng, Nan Duan, Ming Zhou, and Yue Zhang. 2023. Logiqa 2.0—an improved dataset for logical reasoning in natural language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*.

Haoxiong Liu, Yifan Zhang, Yifan Luo, and Andrew Chi-Chih Yao. 2024. Augmenting math word problems via iterative question composing. *arXiv preprint arXiv:2401.09003*.

Liangchen Luo, Zi Lin, Yinxiao Liu, Lei Shu, Yun Zhu, Jingbo Shang, and Lei Meng. 2023. Critique ability of large language models. *arXiv preprint arXiv:2310.04815*.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, and 1 others. 2024. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*.

Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At which training stage does code data help llms reasoning? *arXiv preprint arXiv:2309.16298*.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.

Niklas Muennighoff, Alexander M. Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. 2023. Scaling data-constrained language models. *Preprint*, arXiv:2305.16264.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, and 1 others. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

Robin L. Plackett. 1975. The analysis of permutations. *Applied Statistics*, 24(2):193–202.

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

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. 2022. Self-critiquing models for assisting human evaluators. *arXiv preprint arXiv:2206.05802*.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Yu Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. 2022. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*.

A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*.

Binghai Wang, Runji Lin, Keming Lu, Le Yu, Zhenru Zhang, Fei Huang, Chujie Zheng, Kai Dang, Yang Fan, Xingzhang Ren, An Yang, Binyuan Hui, Dayiheng Liu, Tao Gui, Qi Zhang, Xuanjing Huang, Yu-Gang Jiang, Bowen Yu, Jingren Zhou, and Junyang Lin. 2025. Worldpm: Scaling human preference modeling. *Preprint*, arXiv:2505.10527.

Peiyi Wang, Lei Li, Liang Chen, Feifan Song, Binghuai Lin, Yunbo Cao, Tianyu Liu, and Zhifang Sui. 2023a. Making large language models better reasoners with alignment. *arXiv preprint arXiv:2309.02144*.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Y Wu, and Zhifang Sui. 2023b. Math-shepherd: A label-free step-by-step verifier for llms in mathematical reasoning. *arXiv preprint arXiv:2312.08935*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th Conference on Neural Information Processing Systems (NeurIPS)*.

Haotian Xu. 2023. No train still gain. unleash mathematical reasoning of large language models with monte carlo tree search guided by energy function. *arXiv preprint arXiv:2309.03224*.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, and 1 others. 2024. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

Fei Yu, Anningzhe Gao, and Benyou Wang. 2024. Ovm, outcome-supervised value models for planning in mathematical reasoning. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 858–875.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2023. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*.

Weihao Yu, Zihang Jiang, Yanfei Dong, and Jiashi Feng. 2020. Reclor: A reading comprehension dataset requiring logical reasoning. *arXiv preprint arXiv:2002.04326*.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. 2024a. Generative verifiers: Reward modeling as next-token prediction. *arXiv preprint arXiv:2408.15240*.

Xinlu Zhang, Zhiyu Zoey Chen, Xi Ye, Xianjun Yang, Lichang Chen, William Yang Wang, and Linda Ruth Petzold. 2024b. Unveiling the impact of coding data instruction fine-tuning on large language models reasoning. *arXiv preprint arXiv:2405.20535*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2024. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and 1 others. 2023. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*.

Denny Zhou, Nathanael Sch"arli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and 1 others. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

Xuekai Zhu, Daixuan Cheng, Hengli Li, Kaiyan Zhang, Ermo Hua, Xingtai Lv, Ning Ding, Zhouhan Lin, Zilong Zheng, and Bowen Zhou. 2024. How to synthesize text data without model collapse? *Preprint*, arXiv:2412.14689.

A Hyperparameters and Computational Cost

We outline key hyperparameters and computational cost in this section. In the tables, WSD refers to the warmup-stable-decay learning rate scheduler (Hu et al., 2024), which has the benefit of

Model Size	Number of GPUs	Training Time
Qwen2-1.5B	64 H800	12 hours
Qwen2-7B	128 H800	20 hours

Table 7: Computational cost for CodePMP training across different model sizes.

Hyperparameter	Qwen2-1.5B	Qwen2-7B
epoch	1	1
batch size	1024	1024
learning rate	3e-6	1e-6
lr scheduler	WSD	WSD
warmup ratio	0.03	0.03
decay ratio	0.1	0.1
weight decay	0.1	0.1
max length	1024	1024

Table 8: CodePMP training hyperparameters.

Hyperparameter	Qwen2-1.5B	Qwen2-7B
epoch	1	1
batch size	64	64
learning rate	1e-6	3e-7
lr scheduler	WCD	WCD
warmup ratio	0.03	0.03
weight decay	0	0
max length	1024	1024

Table 9: Mathematical reasoning RM finetuning hyperparameters.

Hyperparameter	Qwen2-1.5B	Qwen2-7B
epoch	1	1
batch size	64	64
learning rate	1e-5	1e-5
lr scheduler	WCD	WCD
warmup ratio	0.25	0.25
weight decay	0	0
max length	1024	1024

Table 10: Logical reasoning RM finetuning hyperparameters.

reducing the time required for scaling law experiments. Specifically, Table 8 lists the hyperparameters for CodePMP training, Table 9 details those for mathematical reasoning RM fine-tuning, Table 10 covers logical reasoning RM fine-tuning, and Table 11 presents the hyperparameters for BON generation.

The computational cost for CodePMP training varies depending on the model size. For CodePMP training with the Qwen2-7B model, we utilized 128 H800 GPUs for approximately 20 hours. For the smaller Qwen2-1.5B model, the training required 64 H800 GPUs for approximately 12 hours. These computational requirements reflect the scalability of our approach across different model sizes while maintaining reasonable training times for the large-scale preference model pretraining. Table 7 summarizes the computational requirements for different model sizes.

Hyperparameter	MetaMath-Mistral-7B
temperature	0.7
top-p	1

Table 11: Best-of-N generation hyperparameters.

Language	Chosen	Rejected
Python	170.0	167.0
Notebook	158.0	155.5
Other Languages	213.2	210.0
Total	**194.5**	**189.9**

Table 12: Average token lengths of responses in the CodePMP dataset by language category.

B CodePMP Dataset Statistics

Table 12 presents the average token lengths of responses in the CodePMP dataset. The similar lengths between chosen and rejected responses (194.5 vs. 189.9 tokens) ensure that response length does not introduce bias in the learning process. The dataset comprises 28 million files totaling 19 billion tokens, with Python (13.1B tokens), Jupyter Notebooks (2.1B tokens), and other languages (3.8B tokens) providing diverse coverage that facilitates model generalization.

C RM Finetuning Dataset

C.1 Mathematical Reasoning

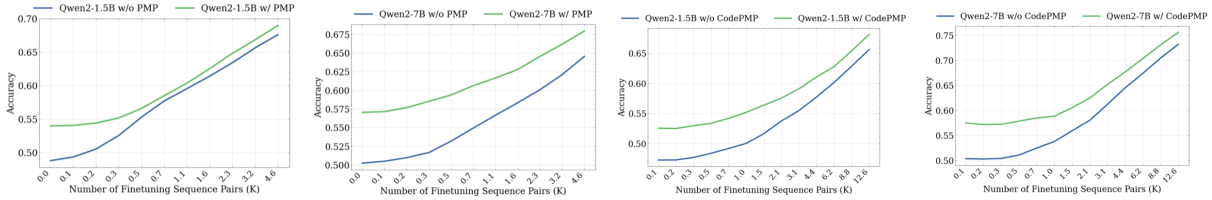
The RM finetuning for mathematical reasoning uses the MathShepherd dataset (Wang et al., 2023b), which contains 444k query-response samples, with some queries having multiple distinct responses. We divide the dataset into a 400k training set and a 44k test set. For RM finetuning, we construct preference pairs by selecting both correct and incorrect responses for the same query. To form the 4.3k test set, we combine one positive and negative sample for each query from the original test set.

We also create two training sets of different sizes: MathShepherd-preference-800k and MathShepherd-preference-40k. The 800k training set is built by combining multiple positive and negative samples for each query in the original training set, resulting in 800k samples. In contrast, the 40k training set randomly selects one positive-negative pair for each query, totaling 40k samples.

C.2 Logical Reasoning

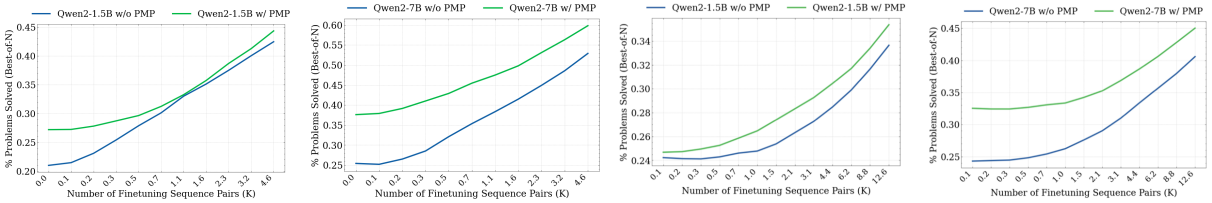
C.2.1 Reclor

Reclor is a human-annotated reading comprehension reasoning dataset, where each sample consists of a passage, a question, and multiple options. To create preference pairs, we combine the correct



(a) MATH-shepherd / 1.5B (b) MATH-shepherd / 7B (c) Reclor+Logiqa / 1.5B (d) Reclor+Logiqa / 7B

Figure 8: Comparison of sample efficiency of RM fine-tuning: Trends of RM accuracy with sample size increases.



(a) MATH-shepherd / 1.5B (b) MATH-shepherd / 7B (c) Reclor+Logiqa / 1.5B (d) Reclor+Logiqa / 7B

Figure 9: Comparison of sample efficiency of RM fine-tuning: Trends of Multi-choice accuracy or Best-of-4 with sample size increases.

and incorrect options for the same question. This process results in a total of 14.5k preference pairs, with 14k pairs used for training and 1.5k for testing, forming the Reclor-preference dataset.

C.2.2 LogiQA2.0

LogiQA 2.0 is a meticulously curated dataset designed for logical reasoning in natural language understanding, focusing on multi-choice question (MCQ). It comprises a substantial dataset of 15,708 instances, each consisting of a passage, a question, and four candidate answers, with the correct answer clearly labeled. The questions and passages translated by professional linguists to ensure clarity and accuracy, while eliminating culturally specific elements. The dataset is annotated with fine-grained logical reasoning types, making it a robust resource for training and evaluating models on complex logical inference tasks.

C.3 CodeUltraFeedback_binarized

CodeUltraFeedback_binarized is a preference dataset in the code domain, consisting of 9.5k preference pairs. We randomly split the dataset, using 90% of the samples for finetuning training and 10% for testing RM accuracy.

D Further Comparisons and Cross-Domain Evaluations

D.1 Comparison with Majority Voting

We compare CodePMP with a majority-voting baseline under the same experimental setup on GSM8K and MATH. Table 13 shows that CodePMP outperforms majority voting, especially on more complex tasks like MATH.

Method	GSM8K	MATH
CodePMP	0.8484	0.41
Majority Voting	0.8453	0.37

Table 13: Comparison of CodePMP and majority voting on GSM8K and MATH.

D.2 Sample Efficiency Improvements on Reclor and LogiQA

We finetune the RM on preference pairs using only Reclor or LogiQA and then evaluate them on their respective test sets. As shown in Figures 8 and 9, PMP demonstrates a clear advantage in sample efficiency, reflected in both RM accuracy and Best-of-N evaluation. The results reveal that even with substantially fewer training samples, reward models initialized with CodePMP achieve comparable or better performance than models trained from scratch with many more samples, highlighting the significant sample efficiency benefits of our approach for logical reasoning tasks.

D.3 Performance on Coding Tasks

We evaluate CodePMP's effectiveness on actual code generation tasks by conducting two types of evaluations: reward model accuracy assessment and code generation evaluation.

First, we assess the reward model's accuracy on the CodeUltraFeedback benchmark, which consists of preference pairs in the code domain. We

Model	PMP	CodeUltraFeedback Accuracy
1.5B	✗	0.6841
	✓	**0.758**
7B	✗	0.6912
	✓	**0.7619**

Table 14: Performance on CodeUltraFeedback benchmark shows that CodePMP improves in-domain code reward modeling.

BoN	Qwen2-7B w/o PMP	Qwen2-7B w/ PMP
N=1	0.7134	0.7134
N=2	0.7317	0.7195
N=4	0.7073	0.7622
N=8	0.6890	0.7683
N=16	0.6951	0.7256
N=32	0.6585	0.7378
N=64	0.6829	0.7134
N=128	0.6707	0.7012
N=256	0.6707	0.7195

Table 15: HumanEval results (Pass@1, 0-shot) for different numbers of sampled solutions N. The generator is deepseek-coder-6.7b-instruct.

fine-tuned Qwen2 models on the CodeUltraFeedback_binarized dataset (8.5k preference pairs), both with and without CodePMP initialization. Table 14 presents the accuracy results across model sizes.

As shown in Table 14, reward models initialized with CodePMP consistently outperform those without PMP initialization on the CodeUltraFeedback benchmark. For the 1.5B model, CodePMP initialization improves accuracy from 0.6841 to 0.758, while for the 7B model, accuracy increases from 0.6912 to 0.7619. These results demonstrate that CodePMP effectively enhances reward models' ability to evaluate code quality.

Beyond reward model evaluation, we also assess whether this improved evaluation capability translates to better code generation outcomes using the HumanEval benchmark. For this evaluation, we used deepseek-coder-6.7b-instruct as the generator and Qwen2-7B as the reward model (RM). We fine-tuned the RM on the same CodeUltraFeedback_binarized dataset, both with and without CodePMP initialization. Table 15 presents Pass@1 (0-shot) results under different N values.

The results in Table 15 indicate that CodePMP initialization provides a generally more stable and higher-accuracy selection mechanism compared to direct training, especially as N varies. For most values of N, the model with CodePMP initialization achieves better Pass@1 scores, with particularly notable improvements at $N = 4$ (0.7622 vs. 0.7073), $N = 8$ (0.7683 vs. 0.6890), and $N = 32$ (0.7378 vs. 0.6585). Without CodePMP, we observe performance degradation at higher N values, while CodePMP-initialized models maintain more consistent performance. This finding is particularly significant since HumanEval evaluates actual code generation rather than just preference prediction, demonstrating that the benefits of CodePMP extend beyond improved preference modeling to better code generation outcomes.

Summary These additional experiments demonstrate that CodePMP:

- Outperforms majority voting in both simpler (GSM8K) and more challenging (MATH) settings.
- Demonstrates significant sample efficiency improvements on logical reasoning tasks (Reclor and LogiQA), with models initialized with CodePMP achieving better performance with fewer training samples.
- Provides more stable and accurate code evaluation on HumanEval, showing benefits for practical code generation tasks.

Thus, CodePMP provides a scalable and effective approach to improving large language models across different domains and tasks.

E Comprehensive Data Diversity Analysis

To validate the quality of our synthetic data, we conducted comprehensive diversity analyses using established methodologies from the research on synthetic text data generation (Zhu et al., 2024). These analyses aim to demonstrate that our synthetic data maintains sufficient diversity while effectively capturing the distributions present in human-generated data.

E.1 N-gram Feature Distribution Analysis

We mapped text n-gram features to fixed hash buckets (100 buckets) and analyzed their distribution patterns to measure lexical diversity. Figures 10 and 11 show the comparison between human-generated data and our synthetic data.

Table 16 presents the density values for unigram and bigram distributions across different data sources.

The distribution graphs show that, compared to human data, our synthetic data has more uniform n-gram distributions, without the concentration peaks common in synthetic data. The density values fur-

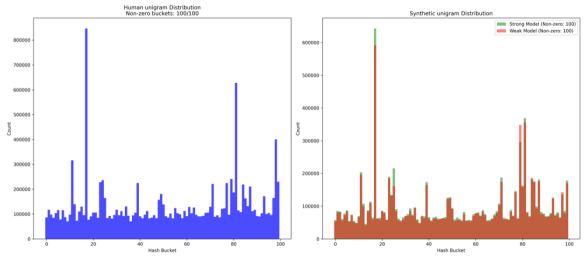


Figure 10: Unigram distribution comparison (left: distribution for human data, right: distribution for synthetic data).

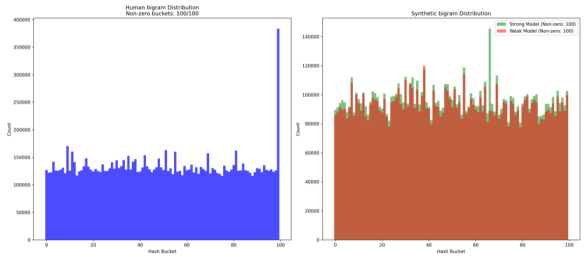


Figure 11: Bigram distribution comparison (left: distribution for human data, right: distribution for synthetic data).

ther quantify this advantage—synthetic text's n-gram density values (Strong Model: 97,653.69, Weak Model: 93,691.69) are significantly lower than human text (134,538.40), demonstrating more balanced distribution across hash buckets.

E.2 Embedding Space Visualization

To further evaluate the semantic diversity of our synthetic data, we mapped semantic features of both human and synthetic data to a 2D space, as shown in Figure 12.

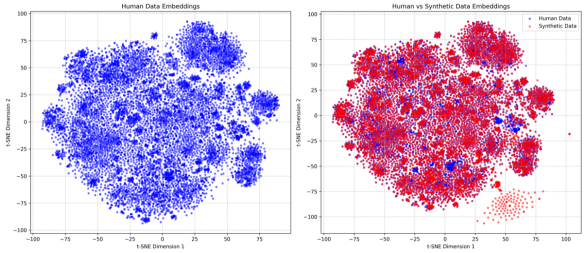


Figure 12: Embedding space visualization (left: distribution for human data, right: distribution for synthetic data).

Both synthetic and human data show wide and dispersed distributions in the embedding space with highly overlapping distribution ranges, indicating our synthetic data captures a similarly broad semantic space as human data.

E.3 KL Divergence Analysis

We quantified the distribution differences between synthetic and human data using KL divergence to evaluate how closely our synthetic data approximates natural distributions. Table 17 presents these results.

Data Source	Unigram	Bigram
Human	134,538.40	133,538.41
Strong Model	97,653.69	96,653.70
Weak Model	93,691.69	92,691.70

Table 16: N-gram density values for human and synthetic data.

N-gram	Human Internal (Bootstrap)	Strong Model vs Human	Weak Model vs Human
1-gram	0.2502	0.4290	0.4631
2-gram	0.6904	1.3500	1.4281
3-gram	1.3012	2.5660	2.6693

Table 17: KL divergence values comparing different data distributions.

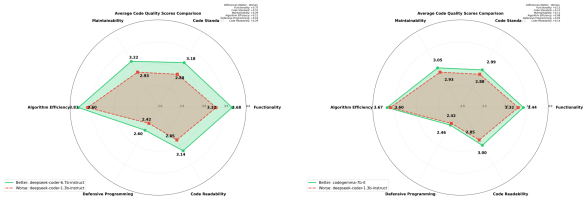
These results demonstrate that the distribution differences between our synthetic data and human data fall within acceptable ranges relative to internal human data variation.

E.4 Comprehensive Validation of Synthetic Preference Data

Our synthetic data generation approach relies on two key assumptions: (1) larger models from the same family produce higher-quality code than smaller ones, and (2) this quality difference creates consistent preference signals suitable for training. We conducted both theoretical and empirical validation to confirm these assumptions.

E.4.1 Validation of Strong-Weak Model Ability Differences

To validate our first assumption, we analyzed ability differences between strong and weak models across various code-related dimensions.



(a) Same model family (b) Different model families

Figure 13: Radar charts showing ability differences between strong and weak LLMs across various dimensions.

Figure 13a provides strong evidence supporting our assumption: when using models from the same architectural family with different parameter counts, the stronger model consistently outperforms the weaker model across all ability dimensions. This uniform superiority ensures that synthetic preference pairs have clear and consistent quality differences, creating reliable signals for training preference models.

For comparison, Figure 13b shows what happens when models from different families are paired. Here, we observe irregular and inconsistent differences, with some dimensions showing negligible gaps or even inversions. Such inconsistencies could potentially introduce noise into the preference signals, undermining training data quality.

E.4.2 External Evaluation of Preference Consistency

Having confirmed the underlying ability differences, we next validated whether these differences translate to consistent preference judgments. We conducted preference annotation experiments using GPT-4o to evaluate the consistency of preferences in our dataset.

The results show that our synthetic CodePMP data achieved a preference consistency rate of 75.12%. This is notably higher than the more costly CodeUltraFeedback preference dataset (71.56%), demonstrating that the preference distinction in our synthetic data (based on our assumption that "larger models generate better code than smaller models") is sufficiently clear and consistent.

This external validation reinforces our first finding - not only do larger models from the same family consistently outperform smaller ones across all dimensions, but this performance gap is readily detectable by strong evaluator models, resulting in consistent preference judgments.

As CodePMP is fundamentally a pretraining process, we deliberately simplified our assumptions to enable scalable preference data creation with minimal additional validation. Our multi-faceted validation approach confirms that this simple yet effective methodology produces high-quality, consistent preference data suitable for large-scale pretraining.

E.5 Source Data Quality and Experimental Validation

Our method achieves excellent diversity due to the high quality of our source data:

- We collected over 130 million code snippets from GitHub, covering all common programming languages and task types on open-source platforms, ensuring breadth and depth in our source data.

Furthermore, our experimental results validate the effectiveness of our synthetic data diversity:

- As shown in Figure 6a, our synthesis strategy outperforms preference pairs constructed directly from source code, indirectly proving that our synthesis process enhances data diversity and quality.

This comprehensive diversity analysis confirms that our synthetic data generation approach produces high-quality, diverse data that effectively captures the distribution characteristics of human-generated code. The balanced distribution patterns and semantic coverage demonstrate that our synthetic data is well-suited for training robust reward models.

F Detailed Implementation of CodePMP

In this section, we provide a detailed overview of the Code Preference Model Pretraining (CodePMP) implementation. The following description illustrates the systematic process of generating and utilizing code-preference pairs for pretraining preference models, which can then be fine-tuned for downstream reasoning tasks.

The algorithm begins with a source code repository, a strong CodeLLM (in our implementation, deepseek-coder-6.7b-instruct), and a weaker CodeLLM (deepseek-coder-1.3b-instruct). First, descriptions are generated for each code snippet in the repository using the strong model. For each description, the strong model generates a high-quality chosen response, while the weaker model generates a less optimal rejected response. These pairs are used to calculate both language modeling loss (on the responses) and reward modeling loss (comparing chosen vs. rejected responses). The final training objective combines these two loss components.

This scalable approach allows for creating millions of preference pairs without expensive human annotation, providing an effective initialization for reward models that will later be fine-tuned on specific reasoning tasks.

G Logical Reasoning Evaluation Examples

We randomly select and present examples from the Reclor test set, which consists of multiple-choice questions based on a given passage. While it is possible to have the model generate additional candidate answers to create a Best-of-N test, it becomes difficult to ensure that the original correct answer remains among the options after introducing new candidates, and to identify the new correct answer. We attempt to use GPT-4o to annotate

Algorithm 1 Code Preference Model Pretraining

Require: Source code repository S,
Strong CodeLLM M_{strong},
Weak CodeLLM M_{weak}
Ensure: Pretrained Model
Input: Source code S
Summarize description D using M_{strong} on S
for each $D_i \in D$ **do**
Generate *Chosen Response* using M_{strong}
Generate *Rejected Response* using M_{weak}
end for
Calculate LM Loss $\mathcal{L}_{\text{LM}}$ on *Response*
Calculate RM Loss $\mathcal{L}_{\text{RM}}$ using *Chosen Response* and *Rejected Response*
Train PMP Model using $\mathcal{L}_{\text{PMP}} = \mathcal{L}_{\text{RM}} + \mathcal{L}_{\text{LM}}$

the correct answers for 32 responses, but the consistency with manual inspection is low, as is the consistency of GPT-4o's own multiple annotations. It can be inferred that the consistency rate would worsen if expanded to 256 responses. Therefore, after careful consideration, we decide to use RM to score only the original four manually annotated answer options, match the top-ranked option with the manually annotated correct answer, and calculate accuracy. In principle, this method is equivalent to the Best-of-4 test.

Table 18: Examples from the Reclor test set, illustrating multiple-choice questions format with passages and questions.

ID	Passage	Question	Ans.
12824	Mayor: When we reorganized the police department, critics claimed it would make police less responsive and lead to more crime. Statistics show an overall decrease in thefts after reorganization.	Which statement most challenges the mayor's argument? (1) Similar reorganizations in other cities led to increased thefts. (2) Unresponsive police reduce theft reporting rates. (3) Critics agree police statistics are reliable. (4) The reorganization saved less money than planned.	2
218	Jupiter is the largest planet with mass 2.5 times that of all other planets combined. Most of Jupiter's 70+ moons are water ice.	What best supports that Jupiter's atmosphere should contain water? (1) Satellites may eventually fall onto planets. (2) Interstellar water exists as gas. (3) Uranus, also a gas giant, contains water ice. (4) Satellites and planets form from the same materials.	3
10376	Lake Dali fish must migrate to river headwaters to breed, though no rivers connect to the sea. Scientists believe these fish originally came from the ocean.	What best explains scientists' belief? (1) Similar fish elsewhere are larger. (2) The fish quickly die in sea/fresh water. (3) Lake Dali was once connected to an ocean-bound river. (4) Fish from Lake Dali survived in far-away lakes.	2
13334	If nuclear waste posed no threat, it could be placed in populated areas. But it is only dumped in sparsely populated regions, suggesting safety concerns.	What would most weaken this argument? (1) Uncertain safety justifies minimal risk placement. (2) Chemical waste is also dumped away from population. (3) Accidents affect fewer people in sparsely populated areas. (4) Remote locations reduce bureaucratic complications.	3