
Self Correction in Lightweight LLMs for Coding Tasks

Team 25 - IDL Final Report

Vidur Sinha
Carnegie Mellon University
vidursin@andrew.cmu.edu

Sudeshna Merugu
Carnegie Mellon University
sudeshnm@andrew.cmu.edu

Manyung Emma Hon
Carnegie Mellon University
mehon@andrew.cmu.edu

Rutvik Joshi
Carnegie Mellon University
rutvikj@andrew.cmu.edu

Abstract

Self-correction is a highly sought-after capability in large language models (LLMs), but modern LLMs have repeatedly been found to struggle with it. Existing approaches to training self-correction often rely on multiple models, a more advanced model, or additional supervision. In this work, we plan to reproduce and experiment with a recently developed reinforcement learning framework for LLM self-correction: Self-Correction via Reinforcement Learning (SCoRe) (1). We will test the performance of this framework on lightweight (1B parameter) LLMs and compare the self-correction capabilities of SCoRe to current prompt-based self-correction techniques. We also experimented with the combination of self-correction prompting strategies as a potential improvement for the SCoRe framework.

1 Motivation and Objective

Self-correction in LLMs is the ability to correct responses over multiple attempts (See Figure 1). This capability is crucial because it would allow models to identify and fix their own errors without human intervention, leading to more reliable AI systems that could be deployed in critical domains such as healthcare, finance, and engineering where error tolerance is minimal. Existing approaches to training self-correction often rely on multiple models, a more advanced model, or additional supervision, making them resource-intensive and impractical for many applications. In this work, we will focus on self-correction for coding tasks, which has applications such as improving AI-driven coding tools, reducing manual debugging, and increasing automation in software development.

In this work, we plan to reproduce and experiment with a recently developed reinforcement learning framework for LLM self-correction: Self-Correction via Reinforcement Learning (SCoRe) (1). Our first objective is to quantitatively evaluate the performance improvement of lightweight LLMs on coding tasks when trained with the SCoRe framework compared to current prompt-based self-correction techniques. Our second objective is to maximize the self-correction capabilities of lightweight LLMs on coding tasks by combining the SCoRe framework with other prompting techniques. This research directly benefits developers of resource-constrained AI applications who need more reliable models without the computational overhead of larger architectures or complex multi-model systems.

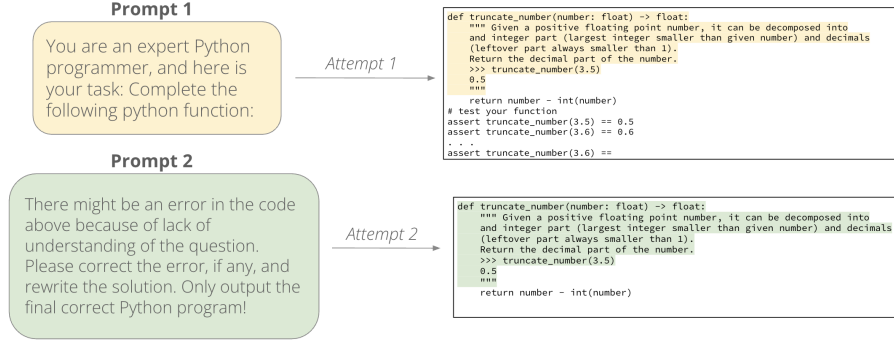


Figure 1: Self Correction Example from the Llama 3.2 1B Model on a HumanEval dataset sample

2 Literature Review and Background

Prior work in self-correction for language models has largely followed two main approaches: prompting-based methods and fine-tuning methods. These approaches have shown significant limitations when applied to both mathematical reasoning and code generation tasks.

The most relevant prior work for our project is STaR (Self-Taught Reasoner) (2), which demonstrates the fundamental challenges in training models to self-correct. STaR attempts to improve model reasoning by fine-tuning on successful reasoning traces. However, as shown in the SCoRe (1) paper, this approach suffers from two critical limitations that our project aims to address:

1. Distribution shift: When trained on offline traces, the model learns to correct errors in a base model’s responses but struggles to correct its own mistakes.
2. Behavior collapse: The model tends to learn superficial editing behaviors rather than meaningful self-correction strategies.

An alternative to SCoRe is ProgCo (3) utilising self-generated programs for error correction, verification and refinement with the main idea that code programs can express more complex logic compared to natural language. However, its effectiveness heavily depends on the accuracy of the generated programs, which are computationally expensive. This limitation makes ProgCo less suitable for our study. Similar drawbacks are seen in CodeRL (4) where an LLM acts as the actor generating code, while the critic evaluates functional correctness.

We aim to implement and evaluate SCoRe’s novel reinforcement learning approach that overcomes these limitations. While STaR and similar methods require multiple models or external supervision, our implementation will follow SCoRe’s single-model approach to achieve self-correction across both mathematical reasoning and code generation tasks. Additionally, we expect to improve these results by incorporating CoT prompts, as we test the performance on lightweight, quantized Language Models.

The Chain-of-Thought (CoT) framework(5) is a reasoning paradigm that enhances the capabilities of LLMs by breaking down complex tasks into a sequence of intermediate steps. In this structure, each step represents a unit of information, allowing the model to reason through each step iteratively. However, CoT with natural language prompts suffers when the prompts underspecify the function that is implemented or when dealing with docstrings that describe a large number of operations and variables (6).

These issues can be mitigated by using Structured Chain-of-Thought (SCoT) (7) which provides structured intermediate reasoning steps for structured source code. This is possible because SCoT prompting explicitly constrains LLMs to think about how to solve requirements from the view of source code and further the performance of LLMs in code generation. Researchers have developed a CoT model generator, Chain Of Thought cOde geNeration (COTTON), which surpasses SCoT especially in lightweight LLMs (8).

3 Methodology

3.1 Overview

To enhance the inference-time self-correction capabilities of code generation models, we build upon the approach introduced in the original SCoRe paper by exploring multiple self-correction prompting strategies. Each prompting method was evaluated in isolation to assess its effectiveness on the HumanEval dataset, as detailed in Section 3.3. The most effective prompting strategy—Structured Chain of Thought—was selected and integrated into the inference process to maximize self-correction performance in our final experiments.

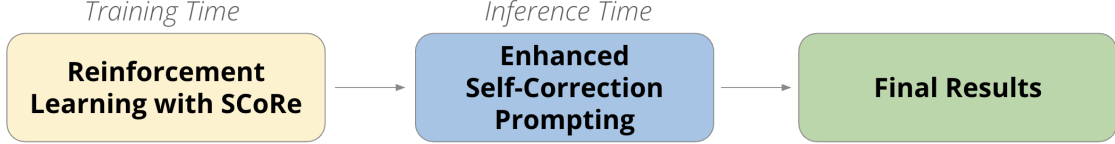


Figure 2: High-Level Methodology

3.2 SCoRe Model Description

SCoRe is a novel approach that trains language models to effectively revise their own answers. Given an initial description or coding prompt (first input), the model produces a response (first output). A self-correcting prompt is then passed to the model as a second input to generate a revised response (second output) based on reflection of its initial answer. The method specifically addresses two key challenges in self-correction: distribution shift (where the first and second attempts become too similar) and behavior collapse (where the model learns to produce perfect first attempts, eliminating the need for correction).

The training process consists of two complementary stages:

Stage I (Initialization for Self-Correction): Unlike standard supervised fine-tuning (SFT), Stage I uses reinforcement learning to establish different behaviors for the first and second attempts. The model is optimized to maintain base model behavior for first attempts through the KL-divergence between the reference model (un-trained) and the training model, and maximize accuracy in second attempts (controlled by β_2 hyperparameter), as seen in Equation 1.

$$\max_{\theta} \mathbb{E}_{x_1, y_1 \sim \pi_{\theta}(\cdot|x), y_2 \sim \pi_{\theta}(\cdot|[x_1, p_1])} [\hat{r}(y_2, \mathbf{y}^*) - \beta_2 D_{KL}(\pi_{\theta}(\cdot|x_1) \parallel \pi_{\text{ref}}(\cdot|x_1))], \quad (1)$$

This initialization creates a natural separation between attempts, establishing a foundation where the model preserves its original response capabilities while developing distinct self-correction mechanisms.

Stage II (Multi-Turn RL with Reward Shaping): Building on this foundation, the model is jointly optimized for both attempts using a carefully designed objective function (See Equation 2) that:

1. Rewards overall correctness in both attempts.
2. Provides a progress-based bonus that encourages meaningful improvements between attempts and penalizes regressions (when the second attempt is worse than the first). This is performed through the following bonus term (with hyper-parameter α) added to the reward for the second attempt in Stage 2: $\alpha \cdot (\hat{r}(y_2, \mathbf{y}^*) - \hat{r}(y_1, \mathbf{y}^*))$.
3. Includes regularization using KL-divergence between the training mode and reference model, controlled by the β_1 hyperparameter.

$$\max_{\theta} \mathbb{E}_{x_1, y_1 \sim \pi_{\theta}(\cdot|x), y_2 \sim \pi_{\theta}(\cdot|[x_1, p_1])} \left[\sum_{i=1}^2 \hat{r}(y_i, \mathbf{y}^*) - \beta_1 D_{KL}(\pi_{\theta}(\cdot|x_i) \parallel \pi_{\text{ref}}(\cdot|x_i)) \right], \quad (2)$$

This reward structure guides the model toward effective revision strategies rather than simply memorizing correct answers or repeating mistakes. The progress-based incentives are particularly important, as they create a learning signal that reinforces the value of the self-correction process itself.

Through this two-stage approach, SCoRe produces language models that can meaningfully reflect on and improve their initial responses, leading to higher overall accuracy and more reliable performance.

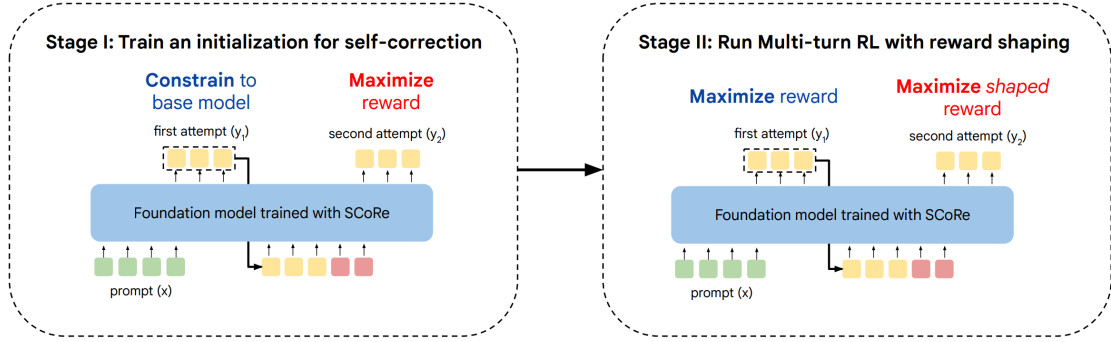


Figure 3: SCoRe Architecture Diagram (1)

3.3 Prompting Approaches

Beyond reinforcement learning, we explored enhanced prompting techniques at inference time to further boost self-correction capabilities. Several prompting strategies were tested to determine their effectiveness individually and in combination with the SCoRe framework.

We experimented with the following prompting approaches:

3.3.1 Baseline Prompt (Used in SCoRe Paper)

This simple prompt suggests that there might be an error due to a misunderstanding of the question, and asks the model to correct and rewrite the solution, outputting only the final corrected code.

Baseline Prompt (SCoRe Paper Style)

“There might be an error in the code below because of lack of understanding of the question. Please correct the error, if any, and rewrite the solution. Only output the final correct Python program!”

3.3.2 Error Identification Prompting

A detailed prompt asking the model to first identify any errors or unhandled edge cases before rewriting the corrected solution.

Error Identification Prompting

“There might be an error in the code below because of lack of understanding of the question. Please identify any errors or edge cases that might not be handled correctly, then rewrite the solution. Only output the final correct Python program!”

3.3.3 Structured Chain of Thought (CoT)

Inspired by chain-of-thought prompting, this structured method includes step by step instructions.

Structured Chain of Thought (CoT)

“You are an expert Python programmer, and here is your task. Before writing any code, think step by step about the following function:

- *What are the inputs and their types?*
- *What is the expected output and its type?*
- *What edge cases should I consider?*
- *What algorithm or approach will solve this efficiently?*

After this analysis, complete the following python function:”

3.3.4 Self-Refine

In this method, the original model (e.g., CodeLlama or Llama) is asked to generate both the feedback and the correction for its initial code output.

Self-Refine

- Here is the original function:
- **[Code generated by the baseline model]**
- Feedback on the function:
- **[Guidance instructions generated by the same model]**
- Please revise the code to fix any issues based on the feedback above. Only output the final corrected Python function. Do not include markdown formatting or explanations. Output only raw Python code.

However, this prompt performed poorly in our experiments. Due to its high computational cost and heavy dependency on the base model’s ability to generate meaningful feedback, we decided not to pursue further experiments using this method.

3.3.5 COTTON (Chain-Of-ThoughtT cOde geNeration)

In this method, instead of using the same baseline model, we use a separate guidance model (version of CodeLlama-7b-Python-hf already fine-tuned specifically for generating guidance prompts) to guide the correction process.

COTTON (Chain-Of-ThoughtT cOde geNeration) Prompt

- Please follow the below instructions to code the Python function:
- **[Guidance instructions generated by a separate model]**

Here, the guidance instructions are generated by querying a separate model:

```
cot = guidance_model.predict(question)
```

This method was found to be more effective compared to Self-Refine, as it decouples the feedback generation from the base model itself, providing clearer, externally guided self-correction instructions. However, it requires access to an additional strong model, introducing additional computational overhead.

3.4 Datasets

Following the SCoRe paper, we will be training the model on the MBPP (Mostly Basic Python Problems) dataset (9) and evaluating on the HumanEval dataset (6). The MBPP dataset, consists of around 1,000 crowd-sourced Python programming problems and is designed to be solvable by entry level programmers. Each problem consists of a task description, code solution, and 3 automated

test cases. The HumanEval dataset, also consists of Python problems, but unlike MBPP, this dataset doesn't expose test cases during inference, making it a more challenging test of true code generation and self-correction abilities.

The HumanEval dataset is part of OpenAI's code evaluation benchmark and is available as an open source repository on github. There are built-in methods to read the JSONL, extract the prompt, test cases, and ideal solution. The MBPP dataset was developed by Google Research and is available on github as well with two versions of the dataset (full and sanitized).

3.5 Evaluation Metrics

To evaluate self-correction performance for at most 1 self-correction (2 possible attempts), we report and analyze the following metrics:

1. **Accuracy@ t_1** – The model's accuracy on the first attempt.
2. **Accuracy@ t_2** – The model's accuracy on the second attempt.
3. $\Delta(t_1, t_2)$ – The net improvement in accuracy between the first and second attempts, indicating the effectiveness of self-correction.
4. $\Delta_{i \rightarrow c}(t_1, t_2)$ – The proportion of problems initially incorrect that become correct on the second attempt, measuring the extent to which self-correction resolves errors.
5. $\Delta_{c \rightarrow i}(t_1, t_2)$ – The proportion of problems initially correct that become incorrect on the second attempt, assessing the model's ability to maintain correct responses.

4 Baselines and Extensions

To ground our experiments and ensure fair comparisons, we selected four models as baselines:

- **Llama 3.2 1B:** A lightweight, general-purpose language model that serves as our non-coding baseline for SCoRe training(10).
- **Deepseek Coder 1.3B:** A specialized, code-focused model pretrained on large repositories of programming data. Its prior exposure to code makes it an ideal coding-specific counterpart for SCoRe's reinforcement-learning self-correction framework. This model is used as our coding baseline model for SCoRe training(11).
- **Llama 3.2 3B:** A mid-sized variant employed to evaluate inference-time prompting strategies. Its increased capacity allows us to gain more useful insights on prompt quality and assess how prompt engineering scales with model size(10).
- **CodeLlama 7B:** An open-source, 7 billion-parameter model fine-tuned extensively on code. We use it as the guidance engine in our COTTON prompting pipeline, owing to its strong code-generation performance(12).

As mentioned in Section 3.3, we used original self-correction prompt from the SCoRe paper as the baseline for our prompt experimentation. This provides a consistent foundation for evaluating the impact of alternative inference-time prompting strategies relative to the base SCoRe framework, which is the primary extension we are proposing in this work.

5 Results and Analysis

5.1 Results Overview

In the following sections, we present and analyze the results of our prompting experiments, our reimplementations of the SCoRe framework, and the integration of SCoRe with our top-performing prompting approach: Structured Chain-of-Thought.

5.2 Prompting Experiments

We explored several prompting strategies beyond our baseline approach to improve the self-correction capabilities of our models on the HumanEval dataset. Table 1 presents a comparison of four distinct prompting methods.

Method	Acc.@t1	Acc.@t2	$\Delta(t1, t2)$	$\Delta^{i \rightarrow c}(t1, t2)$	$\Delta^{c \rightarrow i}(t1, t2)$
Baseline (SCoRe paper prompt)	21.3%	19.9%	-1.5%	0.1%	1.6%
Error Identification Prompting	22.1%	21.0%	-1.1%	0.2%	1.3%
Structured Chain of Thought	24.6%	24.8%	0.2%	0.4%	0.2%
COTTON	24.2%	24.8%	0.6%	0.6%	0.0%

Table 1: Prompting Experimental Results

The baseline approach adopted from the original SCoRe paper achieved 21.3% accuracy on first attempts and 19.9% on second attempts, exhibiting a negative delta of -1.5%. Our Error Identification Prompting strategy showed modest improvements, with 22.1% first-attempt accuracy and 21.0% second-attempt accuracy, reducing the negative delta to -1.1%.

The most significant performance gains came from the Structured Chain of Thought and COTTON approaches. Structured Chain of Thought achieved 24.6% accuracy on first attempts and 24.8% on second attempts, with a positive delta of 0.2%. Similarly, COTTON achieved 24.2% and 24.8% accuracy on first and second attempts respectively, with a more substantial positive delta of 0.6%.

While COTTON demonstrated a higher $\Delta(t1, t2)$ improvement and a stronger $\Delta^{i \rightarrow c}(t1, t2)$ value of 0.6% compared to Structured Chain of Thought’s 0.4%, we ultimately selected Structured Chain of Thought as our primary prompting strategy for integration with SCoRe. This decision was based on several key factors. First, Structured Chain of Thought produced superior first-attempt accuracy (24.6% vs 24.2%), providing a stronger foundation for self-correction. Additionally, COTTON’s implementation complexity presented significant computational challenges. The COTTON approach required using a secondary LLM (CodeLlama-7b) to dynamically generate prompts, substantially increasing computational overhead compared to Structured Chain of Thought’s static template-based approach. In our qualitative analysis, Structured Chain of Thought also generated more interpretable reasoning steps that better aligned with human problem-solving approaches, enhancing the explainability of the model’s solutions while maintaining comparable overall performance to COTTON with less computational expense.

5.3 Baseline vs SCoRe Results

Method	Acc.@t1	Acc.@t2	$\Delta(t1, t2)$	$\Delta^{i \rightarrow c}(t1, t2)$	$\Delta^{c \rightarrow i}(t1, t2)$
Llama 3.2 1B (Baseline)	26.0%	24.0%	-2.0%	1.0%	3.0%
Llama 3.2 1B (SCoRe)	27.0%	25.0%	-2.0%	1.0%	3.0%
Deepseek Coder 1.3B (Baseline)	29.0%	29.0%	0.0%	0.0%	0.0%
Deepseek Coder 1.3B (SCoRe)	34.0%	34.0%	0.0%	0.0%	0.0%

Table 2: Baseline Model Self-correction vs SCoRe-trained Models

As seen in Table 2, SCoRe training yielded marginal gains in overall accuracy but limited impact on self-correction behavior across both models. For Llama 3.2 1B, SCoRe improved Accuracy@t1 from 26.0% to 27.0% and Accuracy@t2 from 24.0% to 25.0%. However, the $\Delta(t1, t2)$ remained static at -2.0%, and the $\Delta^{i \rightarrow c}(t1, t2)$ metric did not show additional improvements compared to the baseline. This indicates that while the model benefited from SCoRe in first-pass solution quality, it did not significantly enhance its error recovery capabilities.

For Deepseek Coder 1.3B, SCoRe improved both Accuracy@t1 and Accuracy@t2 from 29.0% to 34.0%, resulting in a 5% absolute gain. Despite this, $\Delta(t1, t2)$ remained flat at 0.0%, reflecting a

lack of meaningful second-attempt corrections even after reinforcement learning. $\Delta^{i \rightarrow c}$ and $\Delta^{c \rightarrow i}$ metrics also stayed unchanged.

These results suggest three key points:

- **Policy Improvement:** SCoRe primarily improves the model’s initial policy for generating correct outputs rather than inducing effective iterative refinement behaviors.
- **Reward Signal Limitation:** The current reward structure may not provide sufficiently strong or shaped incentives to drive improvements specifically between attempts, particularly in higher-capacity models where initial response quality is already high.
- **Model Capacity Limitations:** We were able to make more improvements between attempts on the higher-parameter Llama model during prompting experimentation. This suggests that these smaller, 1B parameter LLMs may lack the capacity and capability to make meaningful correction between attempts in code generation tasks.

In summary, while SCoRe enhances first-attempt correctness, additional techniques, such as multi-turn revision cycles, progress-based reward shaping, or dynamic prompting, may be necessary to robustly improve $\Delta(t1, t2)$ and $\Delta^{i \rightarrow c}$ in self-correction tasks for lightweight LLMs.

5.4 Final Results (SCoRe + Structured CoT)

Method	Acc.@t1	Acc.@t2	$\Delta(t1, t2)$	$\Delta^{i \rightarrow c}(t1, t2)$	$\Delta^{c \rightarrow i}(t1, t2)$
Llama 3.2 1B (SCoRe)	27.0%	25.0%	-2.0%	1.0%	3.0%
Llama 3.2 1B (SCoRe+CoT)	28.0%	28.0%	0.0%	0.0%	0.0%
Deepseek Coder 1.3B (SCoRe)	34.0%	34.0%	0.0%	0.0%	0.0%
Deepseek Coder 1.3B (SCoRe+CoT)	29.0%	29.0%	0.0%	0.0%	0.0%

Integrating Structured Chain-of-Thought (CoT) prompting with SCoRe training led to mixed outcomes across the evaluated models.

For Llama 3.2 1B, incorporating Structured CoT slightly improved the first-attempt accuracy from 27.0% to 28.0%, while also eliminating the previously observed negative $\Delta(t1, t2)$. The model maintained its performance after self-correction (28.0% at both attempts), indicating greater stability and reduced regression compared to standard SCoRe. However, the proportion of corrected errors ($\Delta^{i \rightarrow c}$) remained at 0.0%, suggesting that while first responses improved, effective error recovery still did not increase.

For Deepseek Coder 1.3B, Structured CoT did not yield improvements. In fact, first-attempt accuracy dropped from 34.0% (SCoRe) to 29.0% (SCoRe+CoT), and $\Delta(t1, t2)$ remained at 0.0%. This indicates that Structured CoT may introduce unnecessary overhead or interfere with models already exhibiting strong first-attempt performance.

Overall, Structured CoT prompting appears helpful for weaker models (like Llama 1B) by boosting initial accuracy and stabilizing self-correction. However, it can be detrimental for stronger models (like Deepseek Coder), potentially due to overspecification or redundancy in the reasoning steps. These results highlight the importance of model-specific tuning when combining inference-time prompting with learned self-correction policies.

6 Future Directions

Our exploration of the SCoRe framework and prompting strategies for self-correcting code generation opens up multiple promising avenues for future work:

Multi-Turn Self-Correction and Dynamic Prompting While our implementation focused on a single revision attempt, future work can extend this to multi-turn self-correction loops where the model iteratively revises its outputs until convergence or a stopping criterion. This could be further enhanced by dynamically adapting the prompting strategy at each step, depending on prior failures or detected reasoning gaps.

Reward Model Refinement Our current reward function prioritizes correctness and penalizes regressions, but it could be expanded to incorporate additional dimensions of quality, such as code efficiency, readability, or alignment with human-generated solutions. Learning reward functions from human preferences or using model-based critics could also enhance feedback fidelity.

Task-Specific Fine-Tuning and Meta-Learning We observed that self-correction performance varies significantly across models. Future efforts may explore meta-learning techniques that allow models to generalize self-correction behavior across task types, or task-specific fine-tuning that better aligns the correction policy with domain-specific error patterns.

Hybrid Self-Correction Frameworks Given the mixed results when combining SCoRe with prompting strategies, future work can explore tighter integration between policy learning and structured reasoning. For example, prompting could be used to guide or scaffold the correction policy, or intermediate reasoning steps could be explicitly supervised within the RL framework.

Scaling to Larger or More Diverse Benchmarks Our experiments were limited to HumanEval and MBPP. Expanding evaluation to broader benchmarks such as CodeContests, HumanEval+, or real-world datasets could better assess generalization and robustness. These more complex settings could also help benchmark long-horizon reasoning and error propagation in multi-step solutions.

Debugging and Interpretability Tools As self-correcting models become more complex, it is important to build tools that explain why a correction was made. Integrating intermediate reasoning traces, decision visualization, or attention tracking can support both debugging and human trust in model behavior.

7 Conclusion

In this work, we reproduced the SCoRe framework for self-correcting code generation and explored whether incorporating inference-time prompting strategies could further enhance its performance. Our replication of the original SCoRe experiments confirmed that policy-based self-correction yields consistent performance gain - even on general-purpose, lightweight models - validating the central claim of the SCoRe paper. We then designed and evaluated three prompting-based self-correction techniques, ultimately selecting structured chain-of-thought for its balance of simplicity and efficiency. While prompting alone provided modest benefits and, when combined with SCoRe, produced mixed results depending on the model, these findings illustrate both the promise and the challenges of hybrid approaches.

The success of SCoRe confirms that policy learning is a robust avenue for self-correction, while the variable behavior of inference-time prompt experimentation highlights the importance of model-specific prompt design. Altogether, our work not only reinforces the value of reinforcement learning for automated code improvement but shows the potential of integrating dynamic reasoning prompts, laying the groundwork for more adaptive, efficient self-correcting coding systems.

8 Acknowledgments

For our SCoRe implementation, we took inspiration from the following GitHub repo: <https://github.com/sanowl/Self-Correcting-LLM-Reinforcement-Learning->. This repository was useful for structural organization of the code, but it had numerous problems including numerous bugs and incorrect reward functions, training implementations, and evaluation implementations. The repo was reworked considerably in our version: <https://github.com/vsinhal124/Score-Scratch/tree/main>.

9 Team Contributions

We have split work up between two subteams, and the work was performed collaboratively within each subteam.

Subteam 1: Vidur and Sudeshna

- Worked on implementing the SCoRe framework, training/evaluating the base models, producing the SCoRe+CoT results.

- Vidur worked on training/evaluating the base Llama model and Sudeshna worked on training/evaluating the DeepSeek Coder model.
- Wrote SCoRe-related sections in the final report.

Subteam 2: Emma and Rutvik

- Performed prompting experiments.
- Emma worked on the Structured CoT, Error Identification, and Baseline prompts. Rutvik worked on Self-Refine and COTTON.
- Wrote prompting-related sections in the Final report.

10 GitHub Repo

Code for SCoRe

Code for Prompting Experimentation

Note: There is a significant amount of code that we used for our experiments which has not been pushed to our repositories. This is due to the power outage in PSC. We can commit/push this code as soon as PSC is available again.

References

- [1] A. Kumar, V. Zhuang, R. Agarwal, Y. Su, J. D. Co-Reyes, A. Singh, K. Baumli, S. Iqbal, C. Bishop, R. Roelofs *et al.*, “Training language models to self-correct via reinforcement learning,” *arXiv preprint arXiv:2409.12917*, 2024.
- [2] E. Zelikman, Y. Wu, J. Mu, and N. D. Goodman, “Star: Bootstrapping reasoning with reasoning,” 2022. [Online]. Available: <https://arxiv.org/abs/2203.14465>
- [3] X. Song, Y. Wu, W. Wang, J. Liu, W. Su, and B. Zheng, “Progco: Program helps self-correction of large language models,” 2025. [Online]. Available: <https://arxiv.org/abs/2501.01264>
- [4] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, “Coderl: Mastering code generation through pretrained models and deep reinforcement learning,” 2022. [Online]. Available: <https://arxiv.org/abs/2207.01780>
- [5] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” 2023. [Online]. Available: <https://arxiv.org/abs/2201.11903>
- [6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” 2021. [Online]. Available: <https://arxiv.org/abs/2107.03374>
- [7] J. Li, G. Li, Y. Li, and Z. Jin, “Structured chain-of-thought prompting for code generation,” 2023. [Online]. Available: <https://arxiv.org/abs/2305.06599>
- [8] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Y. Zhuo, and T. Chen, “Chain-of-thought in neural code generation: From and for lightweight language models,” 2024. [Online]. Available: <https://arxiv.org/abs/2312.05562>
- [9] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, “Program synthesis with large language models,” 2021. [Online]. Available: <https://arxiv.org/abs/2108.07732>
- [10] M. AI, “Llama 3.2: Revolutionizing edge ai and vision with open, customizable models,” <https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/>, 2024, accessed: 2025-04-30.
- [11] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, F. Luo, Y. Xiong, and W. Liang, “Deepseek-coder: When the large language model meets programming – the rise of code intelligence,” 2024. [Online]. Available: <https://arxiv.org/abs/2401.14196>
- [12] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, “Code llama: Open foundation models for code,” 2024.