Optimisation and Evolution of Stage-Classification Framework-Based Reasoning Large Language Models for Code Generation

Xulin Xu*

School of International Education, Neusoft Institute Guangdong, Foshan, Guangdong, China

Abstract. Large language models (LLMs) encounter challenges such as logical errors and non-standard syntax in code generation. Although existing research has proposed various code optimisation approaches, there remains a lack of a unified classification framework to systematise their technical lineage and delineate their applicable boundaries. This paper combines the optimisation phases to systematise and summarise research on reasoning-based LLMs in the domain of code generation, categorising various optimisation methods into three stages: pre-generation optimisation, in-generation optimisation, and post-generation optimisation. Pre-generation optimisation, exemplified by the Self-Planning approach, reduces complexity through macro-level decomposition of coding tasks. In-generative optimisation, exemplified by reinforcement-learning-based test feedback CodeRL and natural-language-based reflective feedback Reflexion, enables real-time feedback and remediation. Post-generation optimisation enhances the quality of the final code generation through three core pathways: code filtering, cooperative evolution, and process rewards. The analysis shows that various methods can enhance code generation quality in specific scenarios, but the differences in computational cost, model dependency and applicability are significant. Constrained by testing benchmarks, model scale, and computational costs, in the future, the focus should be on realistic testing benchmarks, lightweight and efficient optimisation pathways, and multi-stage fusion frameworks to drive the high-quality, practical development of code generation.

1 Introduction

Large language models (LLMs) continue to evolve in the code generation domain, encompassing multiple critical tasks within software engineering. However, the models still encounter numerous practical challenges, such as generating code with logical errors, non-compliant syntax, and variable naming conflicts. Particularly when handling complex business logic or cross-module calls, their reliability diminishes significantly [1, 2]. The problems reflect a flaw in modern LLM code generation, many models rely more on statistical patterns learned from training data rather than real logical inference ability. They

* Corresponding author: xuxulin23@s.nuit.edu.cn

lack a deep understanding of the task's inherent logic, execution environment, and task expectations.

Despite the academic proposals such as the chain of thought and self-reflection, however, there is still lack of a unified framework to classify the technical lineage, intrinsic connections, and applicable scenarios of different approaches. This lack of a classification system confuses researchers and practitioners in knowing the path of technological evolution, and also prevents the selection of optimal technical solutions for specific scenarios.

Therefore, this review takes the optimization stage as the context and systematically analyzes the development and technological innovation of LLM models in the field of code generation from the three optimization stages of pre -, mid -, and post code generation, and post code generation, and compares the performance, application scenarios, and limitations of different optimization methods, providing theoretical basis and practical suggestions for building more reliable, interpretable, and practical code generation LLM.

2 Pre-Generation Optimisation: Planning Principles and Their Evolution

Difference with human developers, LLMs in the generated code usually adopt "direct generation" rather than "think" or "plan". This generation mode neglects the design phase in software engineering, resulting in logical defects in the final generated code. To this end, some researchers have proposed guiding models to construct complete code implementation schemes prior to code generation [3]. This paper categorises such optimisation approaches as 'pre-generation optimisation' methods.

2.1 The enlightenment of the chain of thought

The Chain of Thought (CoT) method was proposed in 2022, with the core idea of guiding the model to generate a complete chain from input through intermediate reasoning steps to output through a small number of thought cases, rather than simply input-output. This approach is a breakthrough in enhancing language model inference and provides a new exploration path for solving complex problems [4]. However, there is a gap between the standard CoT form and the structured program code requirements, resulting in CoT improving the code quality limit.

In order to overcome the problems of traditional CoT, researchers proposed the concept of Structured Chain of Thought (SCoT) in 2023. Basie on CoT, SCOT imposes structured restrictions on intermediate reasoning steps. In order to meet the requirements of the program, it enforces that the model must reason based on the basic structure of the program in the intermediate reasoning steps, and clearly defines the inputs and outputs. It makes the inference process itself more structured, which improves the readability and accuracy of the generated code [5].

Although SCoT realizes the structured reasoning steps on the basis of CoT, its research on code generation still focuses on the micro logical level. In the face of complex code generation tasks that require cross function and multi module, the model still lacks the ability of macro analysis and planning. Researchers have proposed a more macroscopic Self-Planning code generation method (Self-Planning) in response to this limitation.

2.2 Frontier developments in this field: Self-Planning code generation

The Self-Planning, proposed in 2024, is a representative of pre-generation optimization. It proposes a task planning framework consisting of a planning phase and implementation

phase. In the planning stage, through the analysis of the user's natural language description, the model obtains the user's requirements, and transforms the requirements into a high-level structured planning, which defines the implementation steps of the current task, the required modules and the call relationship between the modules. In the generation phase, the model will generate specific code step by step according to this plan [3]. The core of this method is to improve the accuracy and interpretability of code generation by decomposing complex projects into sub tasks that are easy to implement.

The Self-Planning method has many advantages. Firstly, it can improve the accuracy, correctness, readability, and robustness of code generation in complex tasks at a lower cost. Secondly, experiments have been conducted on multiple authoritative code generation benchmarks and various programming languages, and the verification methods have universality and effectiveness. Thirdly, the method is independent of the in-generation and post-generation optimization methods, which can be used in combination and have strong practicality. Fourthly, the research verifies that the "two-stage" framework and "8-shot" promotion are optimal designs through an in-depth ablation study [3].

The experimental results show that the Self-Planning method improves Pass@1 index from 48.1% directly generated to 60.3% on the HumanEval benchmark, which is significantly higher than Few-shot, Code CoT and other methods [3].In summary, Self-Planning achieves strong progress in dealing with more complex code generation problems, and leads the reasoning optimization before generation from the "micro perspective" to the "macro decomposition" stage, which provides a new path for solving more complex real-world programming tasks.

3 Generative Optimisation: The Interaction Between Execution and Feedback

In the pre-generation optimization stage, various methods avoid errors by planning ahead. However, in the actual code generation process, it is still inevitable to generate incorrect or unavailable code due to various problems, such as logical deviation, condition omission, API misuse and so on. In order to ensure that the code can run correctly, researchers have proposed establishing a fast feedback loop mechanism, which can optimize and modify the code in real time based on the signals returned by the actuator. This method deals with code issues during the code generation process and is summarized as the "in-generation optimisation" method in this article.

3.1 Reinforcement learning feedback mechanism

The method of mastering code generation through Pretrained Models and Deep Reinforcement Learning (CodeRL), combining actor critical reinforcement learning mechanism with pre pre-training model and applying it to code generation, which has milestone significance in the field of code generation. In this framework, the pre-training model is regarded as a behavioral network (Actor), and the generated code is run in a unit test environment (Environment), and the test case pass rate is provided as a reward (Reward) after each test. Through this framework, the coderl model can continuously improve the accuracy and quality of generated code through quantitative and objective signals, so as to realize the performance optimization of code generation [6].

The reward signal of the CodeRL model has the characteristics of objectivity and effectiveness, especially suitable for complex code tasks that require a lot of "trial and error" and "correction".The experimental results show that the coderl model performs well on the benchmark of apps and MBPP, and it can achieve Pass@1 The index increased from 1.82%

of CPT-J model to 2.57%, which was achieved on MBPP Pass@80 Up to 63.0%, far surpassing other models of different scales, which fully reflects the advantages of the feedback mechanism of reinforcement learning in the field of code generation [6]. However, this method also has limitations. The characteristics of this mechanism determine that its training process requires a large number of data samples and testing, which increases the computational cost. And its performance depends on the quality and coverage of the unit test set, so it is necessary to consider the selection of the test set.

3.2 Self reflection mechanism

Difference from CodeRL, which relies on a large number of unit test signals, Reflexion proposes a feedback mechanism of natural language reflection to optimize the generated code. This method does not adjust the model parameters, but establishes a "reflection" framework to collect various feedback signals for LLM agents, so as to induce them to learn, repair and make decisions. The core of its optimization is that when the code generated by the agent cannot be executed, it will generate a structured natural language reflection on the error information, with the error information as input. This reflection will be stored by the agent as the basis for diagnosing errors and summarizing lessons, and used to guide the next code generation [7]. This method establishes a virtuous cycle of "action to observation to reflection to action", so that large models can continuously improve their output without fine-tuning to achieve code optimization.

The advantage of Reflexion is that it is flexible and versatile. It is good at handling the rapid debugging and dynamic iteration of competition level programming problems. It can be complementary to the self planning method in 2.2 of this paper. Self planning focuses on pre-generation design and Reflexion on post generation correction. The two can jointly promote the progress of code generation from guessing to upgrading. Experiments show that reflexion has little improvement on the small model, while GPT-4 can achieve an improvement from 80.1% to 91.0% on the HumanEval test set after application, and achieve a historic breakthrough from 7.5% to 15% in the LeetcodeHardGym problem. This reflects that the model has great potential in the field of code generation. However, every method has its limitations. The limitation of this method is that its reflective effect depends on the reasoning ability of the underlying LLM, and the model may be trapped in a solution with a non-optimal local minimum [7].

In summary, both CodeRL and Reflexion represent the idea of introducing feedback loops in the optimization phase of generation. This complements the planning idea of pre-generation optimization and is an important means to improve the quality and degree of code generation.

4 Post generation optimization: closed loop of verification and evolution

Pre-generation planning and in-generation planning optimize the code generation process from two perspectives of code design and real-time error correction, so as to improve the quality of generated code. However, in real life, researchers often hope that the final generated code can be more convincing. This led to the idea of "build now, optimize later" code optimization. The core of this type of idea is to allow the model to generate one or more "candidate codes", and then verify, screen or process these "candidate codes" to ultimately obtain "excellent codes". This article classifies this type of optimization work after the complete code is generated as the "post-generation optimization" method.

4.1 Test screening and verification

Use the generated test generation code (CODET) method on "how to screen the optimum candidate code" this problem is put forward, the method using the same training models of language generation and its core mechanism is the ability to make full use of LLM guide model to generate candidate code, and for the candidate code to generate test cases, Then, a Dual Execution Agreement is adopted for screening, which comprehensively considers the consistency of the output with the use cases and with other sample codes. Final selection in the generated test cases pass rate of the highest in the code as the final code [8]. This method cleverly uses LLM to build a reliable screening mechanism at low cost.

The advantage of the CODET method is that it can use fewer resources and labor costs, increase the coverage of test scenarios, and its framework is simple and effective. As a processing method in the post-optimization stage, it can significantly improve the accuracy of code generation. The experimental results show that the CODET method improves the Pass@1 index of the code-davinci-002 method by 18.8% on the HumanEval benchmark [8]. This reflects that screening based on candidate code can effectively improve the quality and accuracy of generated code in the post-generation stage.

4.2 Unsupervised learning with Co-evolution

Co-evolutionary Unsupervised Learning (CURE) provides another idea of co-evolutionary post-generation optimization. This method trains both the **code generator (Coder)** and the **test case generator (Tester)** simultaneously, enabling them to mutually learn through reinforcement learning, thereby achieving code optimization and upgrading testing mechanisms. In this method, the tester aims to generate tests that can find errors in the coder's code, and the coder aims to pass all tests generated by the tester perfectly [9]. This mechanism can make code generation and verification form a closed loop without supervision, promote the synchronous improvement of code generation and test case quality, and improve the reliability and robustness of code.

The biggest highlight of the cure method is unsupervised and extensible, which makes the code testing work free from manual dependence and can cooperate with self planning to realize the optimization chain from "planning" to "verification" [9]. However, the training process of CURE is complex, and how to balance the optimization of the two LLM models is the most important and difficult point of this method. And in the early stage of low test cases, the optimization effect of this method will be limited.

4.3 Optimization of process rewards

The reward-GRPO method in code generation(Posterior-GRPO) has made innovations in the function reward mechanism. Its research on code generation issues focuses on the "reward cheating" problem, that is, the problem where the model accidentally obtains the correct result through incorrect reasoning. The traditional reinforcement learning (RL) is also reward in the face of this situation, which virtually guides the model to generate code in the wrong direction. To solve this problem, Posterior-GRPO proposed a scheme that rewards the reasoning process of generating code only when the code result is correct [10]. This mechanism requires the model must learn to optimize the internal logic and the final result of the code during training, which forces the correctness, reliability and readability of the code to be improved.

The advantage of the Posterior-GRPO is that it can significantly improve the reasoning quality of the model and reduce logical vulnerabilities. And prevent reward cheating from the source [10]. However, this method needs to use an additional high-quality large reward

model to evaluate the reasoning process, which virtually increases the data annotation cost and training cost.

In summary, the CODET, CURE, and Posterior-GRPO methods respectively represent the three core optimization paths of screening, evolution, and process evaluation in the post-generation optimization stage. These methods reflect that there is still a lot of optimization space after code generation. Whether it is the idea of screening, evolution, or process evaluation, it can significantly improve the code quality, readability, and accuracy after code generation.

5 Horizontal comparison and discussion

In the above sections, the paper systematically combs the representative work and optimization ideas of reasoning LLM in the field of code generation according to the first, middle and last three stages of optimization. In order to further clarify the characteristics and differences of various methods, this chapter will make a macro horizontal comparison and comprehensive discussion of all methods from the three dimensions of performance, applicable scenarios and limitations, so as to provide a clear reference for researchers and practitioners.

5.1 Performance comparison and analysis of each method

This section summarizes the results and optimization rates of various mainstream methods on key benchmarks. It should be noted that the computational costs and usage benchmarks of various methods are different. This article is preferred Pass@1. As a core indicator, it indicates the correctness of code generated at one time. See Formula (1) for details:

$$Pass@1 = E_{problems}\left[1 - \frac{C(n-c,1)}{C(n,1)}\right] \quad (1)$$

If not, select BON (best of n). The baselines are "non optimized versions" of the corresponding methods to ensure that the comparison is relatively fair. The specific results are shown in Table 1.

Table 1. Results and optimization rates of various mainstream methods on key benchmarks.

Method name	Self-Planning[3]	CodeRL [6]	Reflexion [7]	CODET [8]	CURE[9]	Posterior-GRPO[10]
Core model	code-davinci-002(175B)	CodeT5-large(770M)	GPT-4	code-davinci-002(12B)	Qwen2.5-7B	Qwen2.5-Coder-7B
Key Benchmarks	Human Eval	APPS(Intro)	HumanEval (PY)	Human Eval	Human Eval	LiveCodeBench
Core Indicators (%)	Pass@1: 60.3	Pass@1: 6.77	Pass@1: 91.0	Pass@1: 65.8	BoN(16 samples): 51.6	Pass@1: 68.3

Relative optimization ratio	It is 25.4% higher than Direct (48.1)	CE fine-tuning (~2.7) was 150.7% higher than CodeT5	13.6% more than GPT-4 directly generated (80.1)	It is 18.8% higher than zero-sample Direct (47.0)	15.9% more than Qwen2.5-7B-Instruct (35.9)	16.7% improvement over GRPO's code-only reward (58.5)

5.2 Applicable scenarios and selection guidelines

Combined with the characteristics of different optimization methods in the above chapters, Table 2 extracts an applicable scenario of classification and various methods, which can be used as the basis for selecting various methods in practice.

Table 2. Classification stage and applicable scenarios of various methods.

Methods	Optimization phase	Core mechanics	Ideal application scenario	note
Self-Planning [3]	Pre-	Task breakdown	Complex multi-step business logic, algorithm implementation	Rely on large model capabilities
CodeRL[6]	In-	Test feedback	Safety critical scenarios with high quality test suites	Computationally expensive
Reflexion [7]		Reflective repair	Interactive development environment, competition programming	Requires large model support
CODET [8]	Post-	Candidate screening	Quickly filter the best code from a large number of candidate codes	Dependency generation test quality
CURE [9]		co-evolution	Unsupervised environment for code robustness	The training is complex and the initial effect is poor
Posterior-GRPO[10]		Process reward	High reliability requirements, avoid logical holes	Requires reward model, high cost

5.3 Limitations discussion

Although the above methods have made significant breakthroughs in various dimensions such as code readability, reliability, and accuracy, there are still limitations and shortcomings. One limitation of the testing benchmark is that current mainstream datasets such as APPS and HumanEval, tend to focus on algorithmic question types rather than real-world

application environments, which cannot fully simulate the complexity and diversity of actual code generation in real-world tasks. In addition, the benchmark dataset did not include all programming languages, which may result in discrepancies between the evaluation results and actual usage effects. Secondly, there are limitations to various models. Currently, most research relies on the actual capabilities of large-scale LLM models, and there is a significant gap in performance between small-scale parameter models and the large parameter models used in research. This limits the application of low-cost deployment and is not conducive to the healthy development of this field. Secondly, the integration of existing methods with LLM models is not sufficient, and the contextual learning and knowledge graph capabilities in LLM models have not been fully utilized. Thirdly, some methods have high computational costs and complexity, such as CodeRL and CURE, which have highly complex and computationally expensive training processes compared to other methods, greatly increasing the research and application thresholds.

6 Conclusion

This article systematically reviews the development and evolution of inferential LLM models in the field of code generation. By constructing a classification framework with the optimization stage as the core vein, the mainstream optimization methods are divided into three categories: pre-generation planning, in-generation planning and post-generation planning, and the representative optimization methods and their ideas in each category are deeply analyzed. Analysis shows that from the enlightenment of CoT to the macroscopic decomposition of Self-Planning, and then to post-generative optimization techniques such as CodeRL, Reflexion, and CURE based on reinforcement learning, the research in this field presents a trend of continuous deepening and integration from optimizing the internal reasoning ability of the model to introducing external signals for verification. Together, it promotes the field of code generation to be more accurate, more reliable and more practical.

Based on the review analysis in the article, the author believes that future research in the field of code generation can innovate and break through from the following points, in order to promote the solution of common problems in the current field and improve the practicality and value of research. One is to attempt to build evaluation standards that are more in line with real-world code tasks, considering more complex business logic, cross file or device calls, API usage, and wide coverage benchmarks that simulate real-world constraints, while also considering their maintainability and scalability. The second is to explore optimization paths that are lightweight, low-cost, and efficient, or to develop fine-tuning methods with higher parameter efficiency, so that small and medium-sized models can also benefit from optimization methods and lower the threshold for applying technology to reality. The third is to explore the potential application of the core capabilities of LLM models in the field of code generation, so that inferential LLM models can better assist in improving code quality. The fourth is to explore a hybrid optimization mode that combines multiple methods, focusing on different stages of code generation to enhance the ability of developing complex engineering with deductive LLM models.

In summary, the field of code generation will inevitably shift from micro optimization of models to macro development, in order to establish intelligent code generation systems that can meet real-world needs. The technical methods and comparative analysis summarized in this article can provide useful references for this goal.

References

1. N. Huynh, B. Lin, Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. arXiv:2503.01245 (2025)
2. Z. Wang, Z. Zhou, D. Song, et al., Where do large language models fail when generating code? arXiv:2406.08731 (2024)
3. X. Jiang, Y. Dong, L. Wang, et al., Self-planning code generation with large language models. ACM Trans. Softw. Eng. Methodol. 33(7), 1–30 (2024)
4. J. Wei, X. Wang, D. Schuurmans, et al., Chain-of-thought prompting elicits reasoning in large language models. Adv. Neural Inf. Process. Syst. 35, 24824–24837 (2022)
5. F.F. Xu, U. Alon, G. Neubig, et al., A systematic evaluation of large language models of code. in Proceedings of the 6th ACM SIGPLAN Int. Symp. on Machine Programming, 1–10 (2022)
6. H. Le, Y. Wang, A.D. Gotmare, et al., CoderL: Mastering code generation through pretrained models and deep reinforcement learning. Adv. Neural Inf. Process. Syst. 35, 21314–21328 (2022)
7. N. Shinn, F. Cassano, A. Gopinath, et al., Reflexion: Language agents with verbal reinforcement learning. Adv. Neural Inf. Process. Syst. 36, 8634–8652 (2023)
8. B. Chen, F. Zhang, A. Nguyen, et al., CodeT: Code generation with generated tests. arXiv:2207.10397 (2022)
9. Y. Wang, L. Yang, Y. Tian, et al., Co-evolving LLM coder and unit tester via reinforcement learning. arXiv:2506.03136 (2025)
10. L. Fan, Y. Zhang, M. Chen, et al., Posterior-GRPO: Rewarding reasoning processes in code generation. arXiv:2508.05170 (2025)