# Scalability of LLM-Based Multi-Agent Systems for Scientific Code Generation: A Preliminary Study

**Yuru Wang**[1]*, **Kaiyan Zhang**[1]*, **Kai Tian**[1], **Sihang Zeng**[2]
**Xingtai Lv**[1], **Ning Ding**[1], **Biqing Qi**[3]†, **Bowen Zhou**[1,3]†
[1]Tsinghua University, [2]University of Washington, [3]Shanghai AI Laboratory
wangyuru25@mails.tsinghua.edu.cn

## Abstract

Recent studies indicate that LLM-based Multi-Agent Systems (MAS) encounter scalability challenges in complex mathematical problem-solving or coding tasks, exhibiting issues such as inconsistent role adherence and ineffective inter-agent communication. Moreover, the performance advantages of LLM-based MAS over a single agent employing test-time scaling methods (e.g., majority voting) remain marginal. This raises a critical question: *Can LLM-based MAS scale effectively to achieve performance comparable to standalone LLMs or even Large Reasoning Models (LRMs) under optimal test-time compute?* In this paper, we conduct a preliminary investigation into the scalability of LLM-based MAS for scientific code generation. We propose a simple yet scalable two-player framework based on iterative critic-in-the-loop refinement. Our experiments demonstrate that a minimalist actor-critic framework based on DeepSeek-V3 can outperform DeepSeek-R1 under equivalent computational budgets. Surprisingly, more complex frameworks fail to yield significant gains. These findings corroborate recent insights into multi-agent system limitations and highlight the importance of scalable workflows for advancing scientific code generation.

## 1 Introduction

In recent years, LLM-based Multi-Agent Systems (MAS) (Guo et al., 2024) have demonstrated significant potential in complex problem-solving and system coding tasks (Qian et al., 2023; Huang et al., 2023; Qi et al., 2023; Islam et al., 2024; Parmar et al., 2025). In such systems, each agent is assigned a specific role, working collaboratively to achieve predefined objectives, which mirrors human teamwork in real-world scenarios.

However, recent studies (Cemri et al., 2025) reveal that LLM-based MAS often struggle with complex tasks due to issues such as specification ambiguities, inter-agent misalignment, and inadequate task verification. Furthermore, with the growing interest in Test-Time Scaling (TTS) (Zhang et al., 2025b), where performance improves via inference-time compute (e.g., majority voting or best-of-N sampling with reward models) (Snell et al., 2024; Liu et al., 2025; Zhao et al., 2025), the advantages of LLM-based MAS over single-agent systems with TTS diminish under equivalent computational budgets (Zhang et al., 2025a).

Meanwhile, Large Reasoning Models (LRMs) (e.g., DeepSeek-R1 (Guo et al., 2025), OpenAI-o1 (Jaech et al., 2024)) exhibit superior TTS capabilities through extended chain-of-thought reasoning. Yet, these models suffer from high latency and excessive token costs. This raises a critical question: *Can we develop a scalable LLM-based MAS that outperforms standalone LLMs or LRMs with TTS while maintaining efficiency?*

In this paper, we investigate this challenge in scientific code generation (SciCode (Tian et al., 2024)). Unlike mathematical problems, applying TTS (e.g., majority voting) to code generation is inherently difficult. Instead, critique and self-reflection which are strengths of LRMs, play a pivotal role. To address this, we propose a critic-in-the-loop framework to enhance the evaluative capabilities of LLM-based MAS.

Our experiments on the SciCode benchmark demonstrate that a generator-critic framework (using DeepSeek-V3 (Liu et al., 2024)) outperforms standalone DeepSeek-R1 while consuming fewer tokens. Notably, performance improves further with additional critic iterations, suggesting our framework itself serves as an effective TTS strategy. We also explore a three-agent MAS but find no significant gains over the simpler generator-critic approach. Finally, we analyze failure cases to pro-

*Equal contribution.
†Corresponding author.



*Proceedings of The 3rd Workshop on Mathematical Natural Language Processing (MathNLP 2025)*, pages 50–61

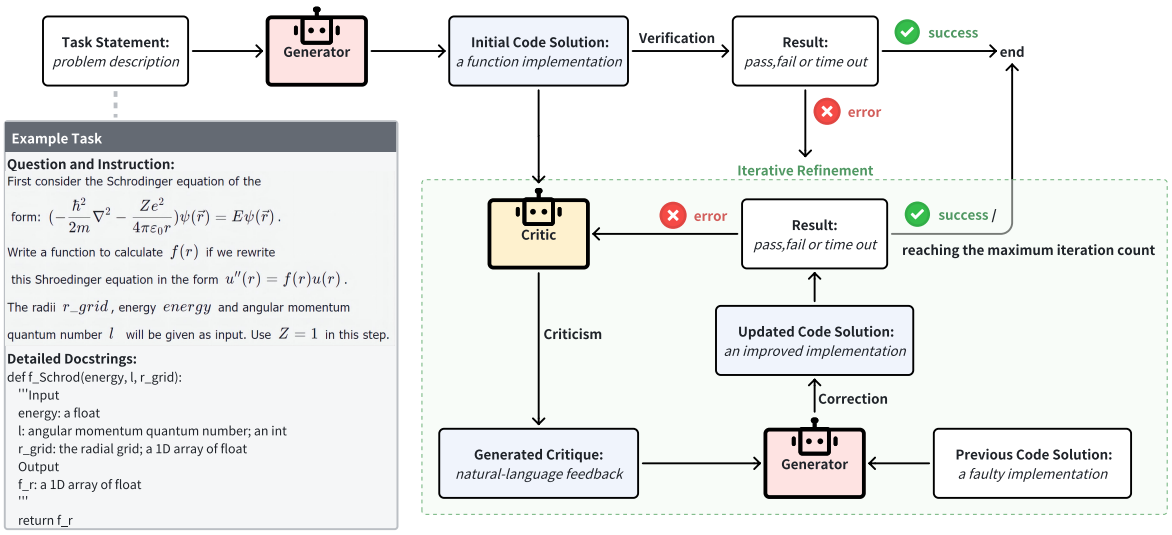



Figure 1: Overview of Generator-Critic Framework.

vide insights for future research.

- We propose a lightweight generator-critic framework that achieves superior performance in scientific code generation compared to LRMs, while reducing token costs by 75%.
- We demonstrate that iterative critic refinement inherently functions as a compute-efficient TTS strategy, unlike traditional voting-based approaches ill-suited for code generation.
- Through ablation studies and failure analysis, we identify key limitations of multi-agent systems (e.g., role confusion in three-agent setups) and provide guidelines for scalable MAS design in the future works.

## 2 Methodology

### 2.1 Iterative Generator-Critic Framework

As illustrated in Figure 1, the framework comprises two specialized agents: **Generator** and a **Critic**.

**Generator**. The Generator initially produces code based on the task description. If the code fails verification, it utilizes feedback containing both the Critic's natural-language critique and the erroneous code to generate an improved version.

**Critic**. The Critic plays a crucial role in the framework by identifying errors in faulty code and providing natural-language feedback. Given a faulty code and a simple failure description (e.g., an error or timeout), it generates nuanced and specific critiques. Unlike scalar rewards, these critiques are more informative, thereby guiding the Generator's revisions more effectively (Shinn et al., 2023).

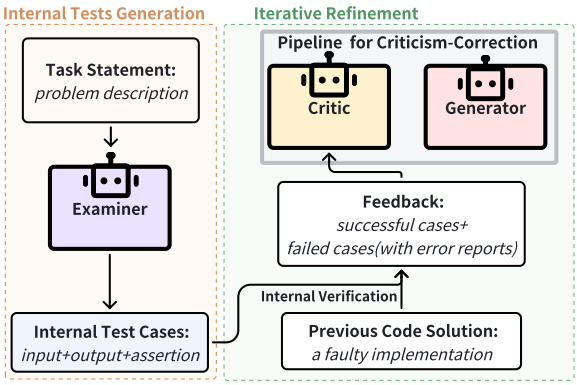



Figure 2: The employment of Examiner.

**Iterative Refinement**. The framework employs a cyclic *criticize* $\rightarrow$ *correct* $\rightarrow$ *criticize* loop (Madaan et al., 2023) to iteratively refine code until a stopping condition (e.g., successful validation or maximum iterations) is met. Formally, given a model $\mathcal{M}$ and an input problem description $x$, the Generator $\mathcal{M}_g$ first produces an initial solution $o_0$,which is then verified by a code interpreter. If verification fails, the Critic and Generator engage in iterative refinement: (1)the Critic $\mathcal{M}_c$ analyzes the previous output $o_{k-1}$ to generate critique $c_k$. (2)The Generator $\mathcal{M}_g$ synthesizes $o_{k-1}$ and $c_k$ to produce an optimized solution $o_k$ that may address previous errors. (3)The new code $o_k$ undergoes revalidation - if successful, the loop terminates; otherwise, the process continues.

| Model | Subproblem Pass@1 | Δ | Main Problem Pass@1 | Δ |
|---|---|---|---|---|
| *Baselines (Single-Agent)* (Tian et al., 2024) | | | | |
| GPT-4o | 25.0 | - | 1.5 | - |
| DeepSeek-V3 | 23.7 | - | 3.1 | - |
| Claude3.5-Sonnet | 26.0 | - | 4.6 | - |
| DeepSeek-R1 | 28.5 | - | 4.6 | - |
| OpenAI-o1-preview | 28.5 | - | 7.7 | - |
| **OpenAI-o3-mini** | **33.3** | - | **9.2** | - |
| GPT-4o (Our) | 22.2 | - | 1.5 | - |
| DeepSeek-V3 (Our) | 25.3 | - | 3.1 | - |
| **DeepSeek-R1 (Our)** | **31.6** | - | **4.6** | - |
| *Generator-Critic (Two-Agent)* (§ 2.1) | | | | |
| *1 iteration* | | | | |
| GPT-4o | 25.0 | ↑ 2.8 | 1.5 | ↑ 0.0 |
| DeepSeek-V3 | 28.5 | ↑ 3.2 | 3.1 | ↑ 0.0 |
| *4 iterations* | | | | |
| GPT-4o | 27.4 | ↑ 5.2 | 4.6 | ↑ 3.1 |
| **DeepSeek-V3** | **32.6** | ↑ 7.3 | **6.2** | ↑ 3.1 |

Table 1: Main Results on Test Set.

### 2.2 Generator-Critic-Examiner Framework

Building upon the iterative multi-agent framework described above, we introduce an **Examiner Agent** to enhance the system's error detection and correction capabilities, as shown in Figure 2.

**Examiner**. Leveraging the chain-of-thought (Wei et al., 2022) reasoning capabilities of large language models, the examiner's primary function is to generate task-specific test cases based on the problem description. Each generated test case contains three essential components: (1) input parameters compliant with the problem requirements, (2) expected outputs representing correct implementation behavior, and (3) assertion statements for automated verification. To improve the output prediction accuracy, we implement a self-consistency mechanism (Wang et al., 2022; Prasad et al., 2025), where multiple predictions are generated for each test input and the final output is determined via majority voting (detailed in Appendix A).

**Test Case Verification Process**. The generated test cases are used to internally verify the code produced by the Generator during the iterative refinement process. The verification results, comprising both successful and failed test cases with corresponding error reports, serve as crucial feedback for the Critic's reflective analysis. If all test cases pass, the iteration terminates and the code is subsequently verified using the gold tests provided by the dataset, with this result determining the final accuracy assessment.

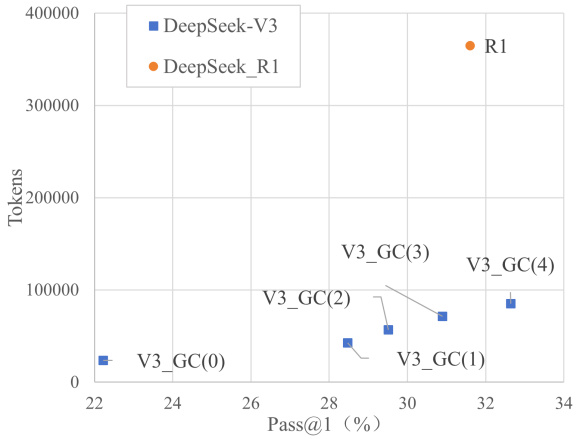



Figure 3: Performance vs. Token Cost between DeepSeek-R1 and iterative Generator-Critic (GC) using DeepSeek-V3. The numbers in parentheses indicate the iteration counts.

## 3 Experiments

### 3.1 Experimental Setup

We evaluate our framework primarily on SciCode (Tian et al., 2024), a scientist-curated coding benchmark comprising 338 subproblems derived from 80 challenging main problems across 16 diverse natural science disciplines. Our implementation builds upon the official codebase [1], with evaluations conducted using GPT-4o and DeepSeek-V1/R1 on both test and validation sets. For baseline comparisons, we incorporate official leaderboard results [2] for GPT-4o, Claude, and OpenAI-o1, while reproducing GPT-4o and DeepSeek-R1/V1 results to ensure consistent evaluation metrics.

### 3.2 Main Results

Table 1 and table 2 present the performance comparison of different methods, which reveal:

**Effectiveness of the Critic Agent.** When employing DeepSeek-v3 as the base model, the Generator-Critic framework achieves a performance improvement (Δ) of 3.2% after one iteration on the test set, which further increases to 7.3% after four iterations. Notably, the Generator-Critic framework consistently outperforms single-agent approach across all evaluated base models, demonstrating the generalizability of the framework. Moreover, the DeepSeek-V3-based framework surpasses the performance of DeepSeek-R1 after four iterations, proving that our multi-agent

[1] https://github.com/scicode-bench/SciCode
[2] https://scicode-bench.github.io/leaderboard/

approach using general-purpose LLMs is highly competitive.

**Enhanced Main Problem Resolution.** The framework particularly excels in solving main problems in SciCode, which require correct solutions for all subproblems. The iterative critique process not only rectifies errors in the current code but also facilitates the resolution of subsequent subproblems - and consequently the main problem - since each subproblem's correctness impacts those that follow. This capability significantly aids in solving the benchmark's most challenging aspects.

**Token Cost Comparison.** Figure 3 compares the token consumption between two approaches: 1) DeepSeek-R1 for code and reasoning outputs, and 2) the iterative Generator-Critic using DeepSeek-V3 for both Generator's code outputs and Critic's critique outputs. The results show that the token consumption of DeepSeek-R1 substantially exceeds that of the Generator-Critic approach, with a substantial difference of 279,605 tokens on the validation set even after four iterations. Despite this, both approaches achieve comparable performance, with our framework even demonstrating superior results (Figure 6). These findings collectively indicate the advantages of the Generator-Critic framework in terms of both efficiency and task performance.

**Number of Iterations.** We examine the efficacy of iterative refinement in both the Generator-Critic and Generator-Critic-Examiner. As shown in Figure 4(a), which presents the pass@1 performance progression on the test set using GPT-4o, iterative refinement consistently improves the framework performance. However, marginal gains diminish as the number of iterations increases, with 5–6 iterations yielding the majority of achievable improvements.

### 3.3 Failure Analysis

Although the Generator-Critic-Examiner outperformed the baselines, it performed worse than the Generator-Critic. This indicates that the Examiner failed to enhance the critic's reflective capabilities. Upon analyzing the test cases generated by the Examiner, we observed substantial inaccuracies in its output predictions. Even with majority voting implemented, the Examiner's predictions remained predominantly incorrect. These errors adversely affected the framework by introducing misleading guidance during refinement, ultimately impairing the efficacy of the criticism mechanism.

| Model | Subproblem Pass@1 | Subproblem Δ | Main Problem Pass@1 | Main Problem Δ |
|---|---|---|---|---|
| *Baselines (Single-Agent)* | | | | |
| GPT-4o | 44.0 | - | 33.3 | - |
| DeepSeek-V3 | 48.0 | - | 46.7 | - |
| DeepSeek-R1 | 50.0 | - | 46.7 | - |
| *Generator-Critic (Two-Agent)* (§ 2.1) | | | | |
| *1 iteration* | | | | |
| GPT-4o | 48.0 | ↑ 4.0 | 33.3 | ↑ 0.0 |
| DeepSeek-V3 | 60.0 | ↑ 12.0 | 40.0 | ↑ 0.0 |
| DeepSeek-R1 | 50.0 | ↑ 0.0 | 46.7 | ↑ 0.0 |
| *4 iterations* | | | | |
| GPT-4o | 50.0 | ↑ 6.0 | 40.0 | ↑ 6.7 |
| DeepSeek-V3 | 62.0 | ↑ 14.0 | 46.7 | ↑ 6.7 |
| DeepSeek-R1 | 56.0 | ↑ 6.0 | 53.3 | ↑ 6.7 |
| *Generator-Critic-Examiner (Three-Agent)* (§ 2.2) | | | | |
| *1 iteration* | | | | |
| GPT-4o | 48.0 | ↑ 4.0 | 33.3 | ↑ 0.0 |
| DeepSeek-V3 | 50.0 | ↑ 2.0 | 40.0 | ↑ 0.0 |
| *4 iterations* | | | | |
| GPT-4o | 48.0 | ↑ 4.0 | 33.3 | ↑ 0.0 |
| DeepSeek-V3 | 54.0 | ↑ 6.0 | 40.0 | ↑ 6.7 |

Table 2: Results on Validation Set.

## 4 Conclusion

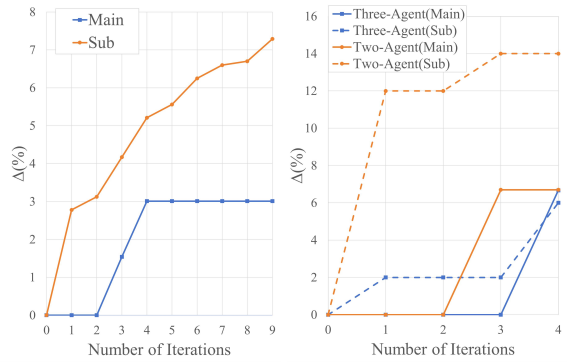
We propose a lightweight generator-critic framework that enhances LLM-based multi-agent systems for scientific code generation. Our approach outperforms standalone Large Reasoning Models while reducing computational costs, demonstrating that iterative critique inherently serves as an efficient test-time scaling strategy. Experiments reveal diminishing returns with complex multi-agent setups, suggesting simplicity is key for scalability. These findings offer practical guidelines for deploying efficient LLM-based systems in resource-constrained scenarios.

## Limitations

Our framework is evaluated solely on scientific code generation (SciCode) tasks. Its effectiveness on other domains (e.g., natural language reasoning or mathematical proof generation) remains unverified, as different problem types may require distinct agent interaction patterns.

The performance gains are demonstrated using specific LLMs (DeepSeek-V3, GPT-4o). Results may vary with smaller or less capable base models, suggesting our approach may be constrained by the underlying model's core capabilities.

While we show computational efficiency gains,




(a) Iterations on Test Set. (b) Iterations on Validation Set.

Figure 4: Iterations in Generator-Critic and Generator-Critic-Examiner.

the critic-in-the-loop approach introduces sequential processing latency. This creates a fundamental tension between token efficiency and real-time responsiveness that may limit deployment in latency-sensitive applications.

## References


Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, and 1 others. 2025. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Runze Liu, Junqi Gao, Jian Zhao, Kaiyan Zhang, Xiu Li, Biqing Qi, Wanli Ouyang, and Bowen Zhou. 2025. Can 1b llm surpass 405b llm? rethinking compute-optimal test-time scaling. *arXiv preprint arXiv:2502.06703*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Mihir Parmar, Xin Liu, Palash Goyal, Yanfei Chen, Long Le, Swaroop Mishra, Hossein Mobahi, Jindong Gu, Zifeng Wang, Hootan Nakhost, and 1 others. 2025. Plangen: A multi-agent framework for generating planning and reasoning trajectories for complex problem solving. *arXiv preprint arXiv:2502.16111*.

Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. 2025. Learning to generate unit tests for automated debugging. *arXiv preprint arXiv:2502.01619*.

Biqing Qi, Kaiyan Zhang, Haoxiang Li, Kai Tian, Sihang Zeng, Zhang-Ren Chen, and Bowen Zhou. 2023. Large language models are zero shot hypothesis proposers. *arXiv preprint arXiv:2311.05965*.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Minyang Tian, Luyu Gao, Shizhuo Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, and 1 others. 2024. Scicode: A research coding benchmark curated by scientists. *Advances in Neural Information Processing Systems*, 37:30624–30650.

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, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Hangfan Zhang, Zhiyao Cui, Xinrun Wang, Qiaosheng Zhang, Zhen Wang, Dinghao Wu, and Shuyue Hu. 2025a. If multi-agent debate is the answer, what is the question? *arXiv preprint arXiv:2502.08788*.

Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, and 1 others. 2025b. A survey on test-time scaling in large language models: What, how, where, and how well? *arXiv preprint arXiv:2503.24235*.

Jian Zhao, Runze Liu, Kaiyan Zhang, Zhimu Zhou, Junqi Gao, Dong Li, Jiafei Lyu, Zhouyi Qian, Biqing Qi, Xiu Li, and 1 others. 2025. Genprm: Scaling test-time compute of process reward models via generative reasoning. *arXiv preprint arXiv:2504.00891*.

# A Details of Methodology

Here, we elaborate on the majority voting mechanism implemented for test cases generation within the Generator-Critic-Examiner framework. First, the Examiner generates input parameters for the function based on the problem description. Next, the agent is invoked multiple times (in our implementation, five repetitions are used) to generate test case outputs through diverse Chain-of-Thought (CoT) reasoning processes. Finally, we tally the frequency of identical output values and select the most common one as the final test case output.This process is shown in Figure 5.

# B System Prompts

The prompts for both the Generator-Critic and the Generator-Critic-Examiner are presented in Tables 3, 4, 5 and 6.

# C Case Study

Here,we provide representative success and failure cases analysis for both the Generator-Critic and the Generator-Critic-examiner.

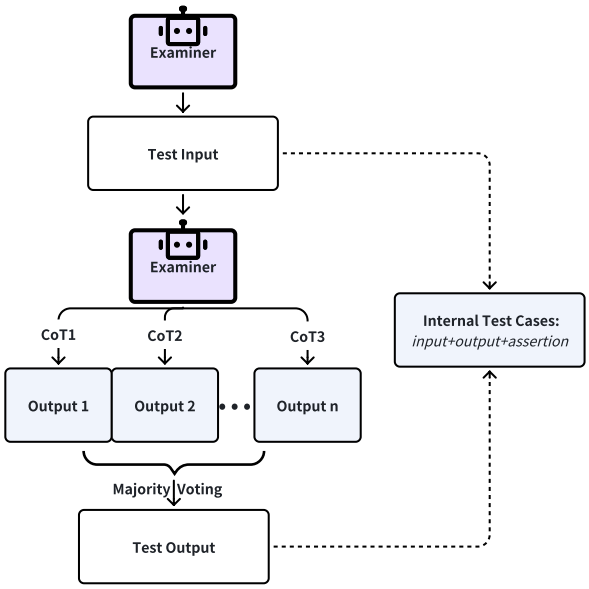



Figure 5: Majority Voting for Test Cases Generation.

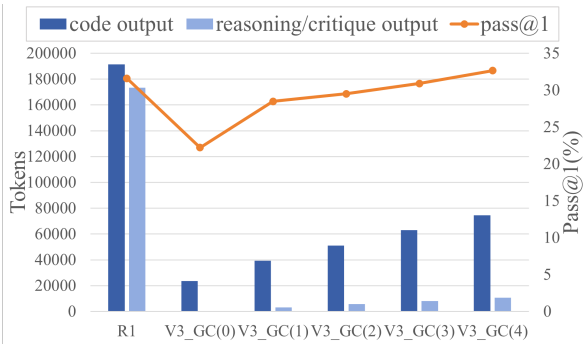



Figure 6: Token cost comparison between DeepSeek-R1 and iterative Generator-Critic (GC) using DeepSeek-V3. The numbers in parentheses indicate the iteration counts.

## C.1 Success Cases

Table 7 presents a successful application of the Generator-Critic framework. In this case,the subproblem $p_i$ was corrected through critique-based optimization, and this correction subsequently revealed the correctness of the following two subproblem $p_{i+1}$ and $p_{i+2}$. This indicates that the initial generated codes for the subsequent subproblems $p_{i+1}$ and $p_{i+2}$ was, in fact, correct; however, due to the error in the preceding subproblem $p_i$, their evaluation resulted in a false failure. By correcting $p_i$ within the Generator-Critic, we were able to verify the true correctness of the $p_{i+1}$ and $p_{i+2}$. This case underscores a key advantage of the Generator-Critic in handling complex, stepwise scientific code generation: accurate evaluation and correction of later steps which require resolving errors in earlier ones.

Table 8 presents the test cases generated by the Examiner for a faulty code which has been successfully corrected. These test cases are particularly valid because the final outputs exhibit a high frequency, indicating their reliability.

### C.2 Failure Cases

Table 9 presents an example in which the errors persist even after reaching the maximum number of iterations in the Generator-Critic framework. Over four iterations, the Critic consistently identified similar error causes, with no significant variation observed.

Table 10 presents test cases demonstrating the Examiner's failure. In these cases, the majority voting mechanism fails to identify a consensus among the predicted outputs, rendering it ineffective.

**EXAMINER IN GENERATOR-CRITIC-EXAMINER FRAMEWORK**

You are an AI coding assistant that can write unique, diverse, and intuitive unit tests for a Python function.
Your job is to generate unit tests that
1.Is valid input based on the function description, i.e., an acceptable input consistent with function description that a correct program should be able to execute.
2. The output enclosed in . and is faithful to the function description, i.e., the output of the unit test is consistent with what a correct program would return.
3. Breaks the code if there is a wrong implementation code based on the function description, i.e., does not execute to the correct output and brings out its mistakes and vulnerabilities.
Provide a reasoning for your answer and identify a general hypothesis or rationale identifying the potential cause of error. Then provide input and output of the unit test consistent with the pattern (hypothesis) you have identified. Note: - that you MUST directly write ALL input arguments of the function in the correct order. Skip writing any names of arguments. -Make sure that hidden associations are satisfied between input arguments. -Make sure that input arguments will not cause a correct program to perform illegal evaluation, such as division by zero encountered in divide or invalid value encountered in scalar add. -You must give the specific value of the outputs. Do not include ellipses or variables without defined specific values in the output. - you MUST enclose the unit test inputs and outputs in. -The inputs and outputs can only be built using the numpy library. -Do not use undefined variables and functions. -Unit tests can only use libraries in dependencies and cannot use other libraries. -Unit tests are independent and cannot use data from each other. -Make sure the logic of the unit test is correct. -You must generate more than four tests.
**## Function Definition:**
{func_sig}
**## Dependencies:**
{dependencies}
Respond strictly in the format below:
**## Hypothesis**
<step-by-step reasoning >
Error Pattern: <an identified pattern of inputs that yields erroneous or incorrect outputs
**## Unit Test X:**
<where X is the unit test number.>
**### Input Arguments**
<step-by-step reasoning for constructing a unit test that fits the error pattern identified above and is valid as per the function description >
Arguments:
{function_header}(< all arguments >)
**### Output**
<step-by-step reasoning for what a correct function_header would execute to based on the function description and your input above. Make sure your data type of the final answer matches the expected output type of the function. Give the specific output directly. Do not use assignment statements and do not provide the code for the calculation process. >
Output:
<your final answer.>
**### Comparison**
<Must use the np.allclose function to compare whether the result of the function matches the output above through the 'assert' statement. The parameter atol of the np.allclose function is set according to the number of digits of the expected output. Write ALL input arguments of the function in the correct order, do not omit input arguments or output. If the function has multiple outputs, compare each output one by one. >
Comparison:
<your code for assert>

Table 3: Prompt for Examiner in Generator-Critic-Examiner.

**CRITIC IN GENERATOR-CRITIC-EXAMINER FRAMEWORK**

You are a Python programming assistant.
You will be given a function implementation and a series of unit tests.The implementation was written under specific requirements and guidance, which are also provided for you. This function implementation is a part of the solution to the complete problem. Implementing it may require calling the code of the preceding steps, which is also provided to you in the requirements and guidance section. Your goal is to write a few sentences to explain why your implementation is wrong as indicated by the tests. You will need this as a hint when you try again later. Only provide the few sentence description in your answer, not the implementation. Only focus on the current implementation, not the preceding steps.
**# Requirements and guidance for writing the current function implementation:**
{prompt}
**# current function implementation:**
{code}
**# preceding steps:**
{previous_code}
**# unit test results:**
{feedback}
**# reflection:**

Table 4: Prompt for Critic in Generator-Critic-Examiner.

**CRITIC IN GENERATOR-CRITIC FRAMEWORK**

You are a Python programming assistant.
You will be given a function implementation and the problem with code(the function implementation with test cases) execution .The implementation was written under specific requirements and guidance, which are also provided for you. This function implementation is a part of the solution to the complete problem. Implementing it may require calling the code of the preceding steps, which is also provided to you. Your goal is to write a few sentences to explain why your implementation is wrong as indicated by the tests. You will need this as a hint when you try again later. Only provide the few sentence description in your answer, not the implementation. Only focus on the current implementation, not the preceding steps.
**# Requirements and guidance for writing the current function implementation:**
{prompt}
**# current function implementation:**
{code}
**# preceding steps:**
{previous_code}
**# problem with code execution:**
{type}
**# reflection:**

Table 5: Prompt for Critic in Generator-Critic.

**GENERATOR IN GENERATOR-CRITIC FRAMEWORK**

You are a Python writing assistant.
You will be given your past function implementation, the problem with code(the function implementation with test cases) execution, and a hint to change the implementation appropriately. The past function implementation was written under the requirements and guidance, your improved implementation should be also under the requirements and guidance. DO NOT write the same implementation as the past function implementation. Write your full implementation.
**# Requirements and guidance for writing the function implementation:**
{prompt}
**# past function implementation:**
{cur_code}
**#problem with code execution:**
{type}
**# hint:**
{reflection}
**# improved implementation:**

Table 6: Prompt for Generator to correct in Generator-Critic.

**CASE STUDY: GENERATOR-CRITIC FRAMEWORK**

**Problem:**
Write down the matrix, $\mathbf{B}$, that transforms $(h, k, l)$ coordinates from the reciprocal lattice system to $(q_x, q_y, q_z)$ coordinates in the right-handed Cartesian system. Let's assume they share an identical origin, with $\hat{\mathbf{x}}^* // \hat{\mathbf{a}}^*$ and $\hat{\mathbf{z}}^* // (\hat{\mathbf{a}}^* \times \hat{\mathbf{b}}^*)$. The direct lattice parameters $(a, b, c, \alpha, \beta, \gamma)$ are given in units of Å and degree. Additionally, we will follow the convention $\mathbf{a_i} \cdot \mathbf{b_j} = \delta_{ij}$, with $\mathbf{a_i}$ and $\mathbf{b_i}$ representing the primitive vectors of crystal lattice and reciprocal lattice respectively Background The reciprocal lattice vectors $\mathbf{b}_i$ are given by:

$$\mathbf{b}_i = \frac{\mathbf{a}_j \times \mathbf{a}_k}{\mathbf{a}_i \cdot (\mathbf{a}_j \times \mathbf{a}_k)}$$

**Critique:**
The implementation is incorrect because it doesn't properly account for the geometric transformation from detector coordinates to lab coordinates. The calculation of $k_{s_{lab}}$ is wrong in how it handles the direction components - it should use trigonometric relationships based on the angles formed by the pixel positions relative to the beam center, not just simple divisions. Additionally, the normalization of the scattered beam vector is incorrect as it doesn't properly maintain the magnitude of 1/wavelength for $k_s$. The current approach mixes coordinate systems incorrectly and doesn't properly represent the scattering geometry.

**Improved Implementation:**

```
def  q_cal(p, b_c, det_d, p_s, wl):
    # Convert all distances to consistent units (angstroms)
    det_d_ang = det_d * 1e7 # mm to angstrom
    p_s_ang = p_s * 1e7 # mm to angstrom
    # Unpack pixel coordinates and beam center
    x, y = p
    xc, yc = b_c
    # Calculate displacement from beam center in detector coordinates (in angstroms)
    dx = (x - xc) * p_s_ang # along -y direction in lab frame
    dy = (y - yc) * p_s_ang # along -z direction in lab frame
    # Calculate the scattering angle components
    r = np.sqrt(det_d_ang**2 + dx**2 + dy**2) # total distance from sample to pixel
    sin_theta_x = dx  r # angle component in x-y plane
    sin_theta_y = dy  r # angle component in x-z plane
    cos_theta = det_d_ang  r # main scattering angle component
    # Incident wavevector (along +x in lab frame)
    k_i = np.array([1/wl, 0, 0])
    # Scattered wavevector (must have magnitude 1wl)
    k_s = (1wl) * np.array([
    cos_theta,
    -sin_theta_x, # negative because dx is along -y
    -sin_theta_y # negative because dy is along -z
    ])
    # Momentum transfer Q = k_s - k_i
    Q = k_s - k_i
    # Return as column vector
    return Q.reshape((3, 1))
```

**Evaluation Result:**
Correct this subproblem successfully, with the next two subsequent subproblems also becoming wright!

Table 7: Success Example of Generator-Critic.

**CASE STUDY: GENERATOR-CRITIC-EXAMINER FRAMEWORK**

**Unit Test 1:**
**# Input Arguments**
orthogonalize(np.array([[1, 0, 1], [1, 1, 0], [0, 1, 1]], dtype=float))

**# Predicted Outputs by different CoT**
**Output 1:** np.array([[0.70710678, -0.40824829, 0.57735027], [0.70710678, 0.40824829, -0.57735027], [0., 0.81649658, 0.57735027]])
**Output 2:** np.array([[ 0.70710678, -0.40824829, 0.57735027], [ 0.70710678, 0.40824829, -0.57735027],[ 0. , 0.81649658, 0.57735027]])
**Output 3:** np.array([[ 0.70710678, -0.40824829, 0.57735027],[ 0.70710678, 0.40824829, -0.57735027],[ 0. , 0.81649658, 0.57735027]])
**Output 4:** np.array([[ 0.70710678, -0.40824829, 0.57735027],[ 0.70710678, 0.40824829, 0.57735027],[ 0. , 0.81649658, 0.57735027]])
**Output 5:** np.array([[ 0.70710678, -0.40824829, 0.57735027],[ 0.70710678, 0.40824829, -0.57735027],[ 0. , 0.81649658, 0.57735027]])

**# Majority Output and the Frequency**
**Final Output:** np.array([[0.70710678, -0.40824829, 0.57735027], [0.70710678, 0.40824829, -0.57735027], [0., 0.81649658, 0.57735027]])
**Frequency:** 80%

**Unit Test 2:**
**# Input Arguments**
orthogonalize(np.array([[1, 1], [0, 1]], dtype=float))

**# Predicted Outputs by different CoT**
**Output 1:** np.array([[1., 0.], [0., 1.]])
**Output 2:** np.array([[1., 0.], [-0., 1.]], dtype=float)
**Output 3:** np.array([[1., 0.], [0., 1.]])
**Output 4:** np.array([[1., 0.], [0., 1.]])
**Output 5:** np.array([[1., 0.], [0., 1.]])

**# Majority Output and the Frequency**
**Final Output:**np.array([[1., 0.], [0., 1.]])
**Frequency:** 100%

**Unit Test 3:**
**# Input Arguments**
orthogonalize(np.array([[2, 0], [0, 3]], dtype=float))

**# Predicted Outputs by different CoT**
**Output 1:** np.array([[1., 0.], [0., 1.]])
**Output 2:** np.array([[1., 0.], [-0., 1.]])
**Output 3:** np.array([[1., 0.], [0., 1.]])
**Output 4:** np.array([[1., 0.], [0., 1.]])
**Output 5:** np.array([[1., 0.], [0., 1.]], dtype=float)

**# Majority Output and the Frequency**
**Final Output:**np.array([[1., 0.], [0., 1.]])
**Frequency:** 100%

Table 8: Success Example of Examiner.

**CASE STUDY: GENERATOR-CRITIC FRAMEWORK**

**Problem:**
Now write a function to analyze any damped, driven pendulum system to understand its dynamic behavior under various conditions. Your function should:
Sweep different timesteps to find the optimized timestep that balances accuracy and time efficiency.
2. Output the trajectory
The combined metric for finding the optimized time should penalize higher computational times while rewarding lower errors. To better reflect this, we can use a combined metric such as:

$$\text{Metric} = \text{GTE} \times \sqrt{\text{Time}}$$

Global Truncation Error (GTE) is estimated using the step-doubling method. This way, longer computation times will have a more significant penalty, while still prioritizing low errors.

**Critique:** 1 iteration
The implementation is incorrect because it does not properly handle the step-doubling method for estimating the Global Truncation Error (GTE). Specifically, the error estimation is only comparing the final state of the trajectory with the half-timestep trajectory, which does not accurately represent the cumulative error over the entire time span. Additionally, the combined metric calculation does not account for potential numerical instabilities or edge cases, such as when the timestep is too large or too small. The current approach may lead to suboptimal timestep selection due to these inaccuracies in error estimation and metric computation.

**Critique:** 2 iterations
The implementation fails because the error calculation in the step-doubling method is incorrect. The current approach compares trajectories of different lengths by taking every other point from the finer trajectory, but this doesn't properly account for the time alignment between the two solutions. Additionally, the error normalization (dividing by sqrt(min_length)) may not appropriately scale the error for comparison across different timesteps. The metric calculation also doesn't properly handle cases where the trajectories have different time points due to different step sizes.

**Critique:** 3 iterations
The current implementation fails because it doesn't properly handle the time adjustment when calculating trajectories with different timesteps. The main issues are: 1) The error calculation compares trajectories of potentially different lengths due to integer rounding of steps, 2) The actual_dt adjustment isn't properly synchronized between the full and half-step trajectories, and 3) The metric calculation doesn't account for cases where trajectories become significantly misaligned in time due to step size adjustments. These issues lead to incorrect error estimates and suboptimal timestep selection.

**Critique:** 4 iterations
The implementation is incorrect because it does not properly handle the time points when comparing trajectories with different step sizes. The error calculation assumes the trajectories can be directly compared by interpolation, but the interpolation may not accurately capture the error due to the non-linear nature of the pendulum's motion. Additionally, the combined metric (GTE $\times \sqrt{}$ Time) may not be correctly weighted, leading to suboptimal timestep selection. The function also does not account for edge cases where the timestep does not evenly divide the total time interval, potentially causing inaccuracies in the final trajectory.

**Final Implementation:**

```
def pendulum_analysis(g, L, beta, A, alpha, initial_state, t0, tf, min_dt, max_dt, num_timesteps):
    timesteps = np.logspace(np.log10(min_dt), np.log10(max_dt), num_timesteps)
    best_metric = float('inf')
    optimized_trajectory = None
    for dt in timesteps:
        n_steps = int(np.ceil((tf - t0) / dt))
        actual_dt = (tf - t0) / n_steps # Adjusted dt to exactly reach tf
        start_time = time.time()
[...]
        if metric < best_metric:
            best_metric = metric
            optimized_trajectory = trajectory
    return optimized_trajectory
```

**Evaluation Result:**
Still wrong.

Table 9: Failure Example of Generator-Critic.

**CASE STUDY: GENERATOR-CRITIC-EXAMINER FRAMEWORK**

**Unit Test 1:**
**# Input Arguments**
sum_real_cross( np.array([1.0, -1.0, 2.0]), np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, 1.0, 0.0]]), np.array([[0.0, 0.0, 0.5], [0.0, 0.5, 0.0]]), np.array([[1, 1, 1], [-1, -1, -1]]), 0.5)

**# Predicted Outputs by different CoT**
**Output 1:** -0.577102
**Output 2:** 3.142
**Output 3:** 0.1
**Output 4:** 2.0
**Output 5:** 0.0

**# Majority Output and the Frequency**
**Final Output:** -0.577102
**Frequency:** 20%

**Unit Test 2:**
**# Input Arguments**

sum_real_cross( np.array([1.0, 1.0]), np.array([[1.0, 0.0, 0.0], [2.0, 0.0, 0.0]]), np.array([[0.5, 0.5, 0.5]]), np.array([[0, 0, 0]]), 0.2 )

**# Predicted Outputs by different CoT**
**Output 1:** 0.493671
**Output 2:** 0.0
**Output 3:** 0.5
**Output 4:** 0.9
**Output 5:** 1.0

**# Majority Output and the Frequency**
**Final Output:** 0.493671
**Frequency:** 20%

**Unit Test 3:**
**# Input Arguments**
sum_real_cross( np.array([1.0, -1.0]), np.array([[1.0, 1.0, 1.0], [0.0, 0.0, 0.0]]), np.array([[1.0, 1.0, 1.0], [0.0, 0.0, 0.0]]), np.array([[0, 0, 0]]), 0.1)

**# Predicted Outputs by different CoT**
**Output 1:** -0.999999
**Output 2:** 4.107857649106695
**Output 3:** -0.6065306597
**Output 4:** -0.1
**Output 5:** -0.427547

**# Majority Output and the Frequency**
**Final Output:**np.array([[1., 0.], [0., 1.]])
**Frequency:** 20%

Table 10: Failure Example of Examiner.