Compare-and-Correct: Test-Time Scaling for Scientific Coding Agents *without* Gold Verification Signals

Anonymous ACL submission

Abstract

Scientific coding tasks often require navigating vast, uncertain solution spaces, where initial hypotheses may be incomplete, flawed, or contradictory. A key challenge in automating scientific coding tasks is the lack of clear-cut success signals, such as gold labels or unit tests that are common in traditional programming or supervised learning tasks. In scientific domains, correctness is often context-dependent, diverse in form, and rarely captured by a single metric. This makes it difficult for agents to determine when a solution is sufficient or how to refine it. We introduce `Compare-and-Correct` (`C&C`), a verifier-guided agent framework for scalable test-time trajectory exploration. Instead of relying on single-pass inference, `C&C` leverages test-time compute scaling by generating a diverse set of candidate solutions and then iteratively refining them via self-debugging and self-improvement mechanisms. An Elo Rating-based verifier ranks candidates by relative quality, guiding the agent to backtrack, correct, and converge on the most promising solutions without relying on explicit success criteria. We demonstrate `C&C`'s effectiveness across a range of tasks including machine learning engineering and visualization on ScienceAgentBench. Experiments show that `C&C` significantly outperforms direct prompting, prior agents like OpenHands and Self-Debug, and alternative verifiers such as random selection and LLM-as-a-Judge. These results confirm the strength of our agent design and verification approach. [1]

1 Introduction

Large language models (LLMs) for science are rapidly reshaping the landscape of scientific discovery by enabling LLMs to reason, analyze, and solve problems across domains such as chemistry, biology, physics, and data science (Yu et al., 2024, 2025; Tang et al., 2024; Bhattacharya et al., 2024; Ali-Dib and Menou, 2024; Hong et al., 2024; Chen et al., 2025). A wide spectrum of tasks, including molecule design (Bhattacharya et al., 2024), physics simulation (Ali-Dib and Menou, 2024), hypothesis formulation (Jin et al., 2024), and experiment-guided machine learning and data analysis (Chan et al., 2025; Nathani et al., 2025) pose unique challenges in terms of multi-step reasoning, domain-specific code generation, and integration with structured scientific data. To systematically study and advance these capabilities, recent benchmarks such as ScienceAgentBench (Chen et al., 2025), DiscoveryBench (Majumder et al.), SciCode (Tian et al., 2024), and BixBench (Mitchener et al., 2025) formulate scientific discovery and numerical calculation problems as code generation tasks and focus on evaluating advanced models and agents across real-world scientific problems. These benchmarks mirror the complexity of real-world science and reveal the growing potential of LLMs not merely as language models, but as AI co-scientists capable of accelerating and expanding the frontiers of scientific inquiry.

To solve those scientific tasks, many rely on direct prompting to obtain programs without special agent design (Tian et al., 2024). While some agents are emerging, most of them face at least one of the following limitations in their ability to generalize and robustly support scientific coding tasks: (1) only leverage a linear problem-solving strategy and lack structured reasoning and solution space exploration, which prevents full utilization of the models' capabilities (Wang et al., 2024; Chen et al., 2024). (2) Rely on the gold verification signals such as clear-cut success metrics of the problem to provide feedback for further refinement and lacks a more general verifier to select the best solution during the exploration (Jiang et al., 2025), which often does not hold for real-world scientific settings and restricts its scope of generalization to more problems. These limitations highlight the need for science

[1] Our source code will be available upon acceptance.

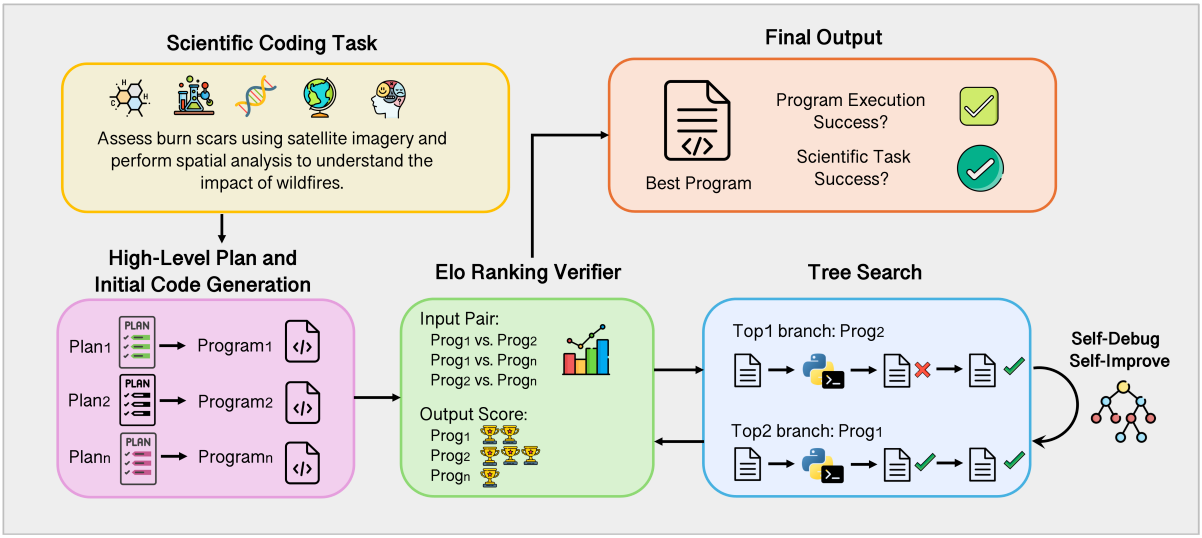

Figure 1: An overview of our agent framework: `Compare-and-Correct (C&C)`.

agents to wisely plan, reason, explore and verify autonomously in realistic scientific tasks.

Existing work such as PlanSearch (Wang et al., 2025) and CodeMonkeys (Ehrlich et al., 2025) show that as the number of generated solutions increases, the fraction of problems in a dataset that are successfully solved by at least one candidate often grows approximately log-linearly. This test-time compute scaling effect of generating more candidate solutions can significantly improve the overall success solution coverage such as Pass@K in coding tasks. Under this background, we seek to answer two central questions: (1) How can we leverage test-time scaling to explore a broader space of potential solutions for scientific coding tasks? (2) How can we design an effective verifier that can reliably select the most plausible solution from a set of search trajectories, especially when those solutions in the trajectories vary in quality, completeness, and logical soundness?

A key challenge in scientific coding tasks is the absence of clear-cut success signals such as unit test cases or gold labels, which are commonly available in traditional programming tasks or supervised MLE benchmarks like Kaggle. In scientific domains, correctness is often diverse, context-dependent, and not easily captured by a unified format such as a single accuracy metric or unit test pass rate. This makes it difficult for agents to know when a solution is good enough or how to improve upon it.

To address this, we propose `C&C`, a novel verifier-guided agent framework that integrates test-time scaling for trajectory exploration. As Figure 1 shows, our framework embraces test-time diversification, which generates a broad set of initial candidate solutions to create a meaningful basis for comparison. We then use an Elo Rating-based verifier to identify the most promising candidate based on relative quality, enabling targeted self-debugging and self-improvement. This design allows the agent to adaptively search, compare, and improve solutions without relying on hard-coded metrics, making it well-suited for open-ended, complex and weakly supervised scientific coding tasks.

We conduct a detailed component-wise analysis to better understand the contribution of each design choice. Through targeted ablations and controlled experiments, we evaluate the impact of initial solution size, self-improvement mechanism, and verifier strategies. This analysis reveals how each module contributes to end-to-end performance, and demonstrates that the combination of diverse solution exploration and strategic verifier-driven selection is key to effective scientific coding tasks.

In summary, our contributions are three-fold:

- We propose a novel agent framework for scientific coding tasks that explores the solution space more thoroughly through verifier-guided tree search, enabling iterative refinement and evaluation of candidate programs without relying on gold verification signals.
- We conduct a detailed analysis of each component to better understand how it affects the performance of our agent framework.

- Our results show that the success rate of C&C on ScienceAgentBench achieves a significant gain over other baselines with reasonable cost. It exceeds the best baseline agent Self-Debug with a 24% improvement for SR, and achieves 22.9% improvement over OpenHands for SR while reducing 53.2% cost.

2 Agent Framework

2.1 Background and Challenges

With the rapid development of LLMs, more and more increasingly complex problems can now be addressed via direct prompting (Tian et al., 2024). However, this method often lacks structured reasoning and struggles with complex reasoning with intricate dependencies across steps. Agent-based frameworks like OpenHands (Wang et al., 2024) improve upon this by generating and executing partial code iteratively, using intermediate observations to inform subsequent actions. This design enables LLMs to reason and act more dynamically, facilitating more effective problem-solving and greater flexibility. The basic ReAct (Yao et al., 2023) (Reasoning + Acting) prompting used in OpenHands encourages step-by-step thinking and external tool usage, making it well-suited for tasks requiring multi-step reasoning. Nevertheless, these approaches still follow a linear reasoning process and may not capture deeper logical dependencies. Self-debugging agents (Chen et al., 2024) further improve execution success through iterative retries and leverage the execution feedback generated by code interpreter, but still fail to address deeper semantic errors or flaws in scientific logic. While AIDE (Jiang et al., 2025) introduces tree search for program-level exploration for automated code refinement, it relies on the clear-cut success metrics in tasks as the explicit feedback and keep running agents to attempt more if the solution doesn't meet the metric requirement. However, in scientific tasks, such success criteria is often not available so that the assumptions in AIDE's agent rarely hold in real-world scientific domains.

2.2 Design Motivation

Scientific discovery is rarely linear. It thrives on exploration, revision, and reflection. Yet, most current reasoning agents make shallow, one-shot decisions, lacking the capacity to backtrack, revise, or evaluate solutions strategically. C&C is designed to overcome these limitations by treating scientific reasoning as a trajectory-driven process, where multiple diverse paths are explored, evaluated, and refined. Not all paths are equal: some lead to breakthroughs, others to dead ends. C&C introduces a verifier-guided tree search that compares, scores and expands solution trajectories based on Elo Rating. This allows the agent to strategically prioritize promising directions and enables dynamic solution expansion, self-debugging, and self-refinement, mimicking the way human scientists iterate toward insight.

C&C is inspired by human-like reasoning processes, where initial rough ideas or high-level plans are generated first, followed by iterative debugging and refinement. This approach allows the agent to progressively improve its solutions through self-correction. Our agent mainly explores the potential solution through tree search. The inclusion of a backtracking mechanism during search enables the agent to revisit earlier reasoning steps and reconsider alternative paths when necessary. A dedicated verifier guides the agent's next actions: determining which branches to explore, how to prioritize the search, and which solutions warrant further refinement, thereby enabling more deliberate and adaptive problem-solving.

2.3 Components

Initial Planning and Solution Generation. Existing work such as PlanSearch (Wang et al., 2025) and CodeMonkeys (Ehrlich et al., 2025) show that as the number of generated solutions increases, the fraction of problems in a dataset that are successfully solved by at least one candidate often grows approximately log-linearly. This test-time compute scaling effect of generating more candidate solutions can significantly improve the overall success solution coverage such as Pass@K in coding tasks. Hence, we follow (Jiang et al., 2025) to generate a rich pool of diverse solution candidates to give more starting points to increase the agents' exploration success. As Figure 1 shows, we first prompt LLM to generate multiple high-level plans, and ensure that previously generated plans are visible to the model to avoid repeated plans. Then for each plan, the LLM generates a corresponding program as a candidate solution. This approach unlocks hidden insights that would be missed by deterministic one-pass inference alone.

Self-Debug for On-the-Fly Error Correction. Scientific coding tasks require the agent to produce executable code. C&C incorporates a self-debugging

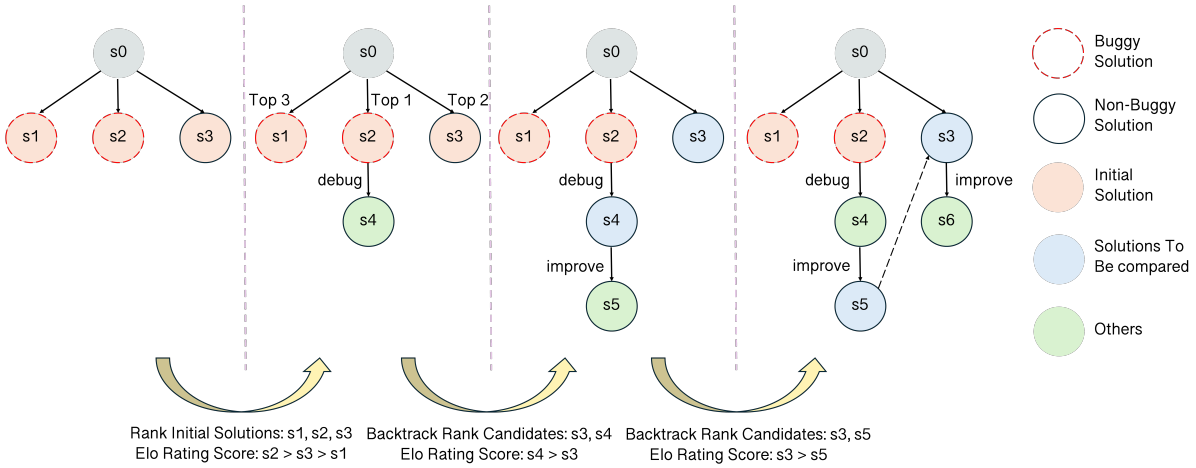

Figure 2: An overview of Elo Rating verifier in tree search.

mechanism to leverage code interpreter to detect and repair bugs during tree search. This reflective capability enables the agent to revise faulty steps without discarding entire trajectories.

Iterative Self-Improve through Reflective Reasoning. Reasoning is not just about fixing mistakes, but about getting better with each step. C&C employs iterative self-refinement by prompting the model to identify a specific refinement point within a selected frontier solution, based on the task description. This process mirrors how humans iteratively refine solutions, progressively improving them toward correctness and completeness.

Verifier for Frontier Selection. Effective trajectory selection is critical to the success of C&C. We implement a suite of frontier selection mechanisms to explore which selection choice is more effective to evaluate and rank candidate solutions throughout the exploration. The same mechanism is employed both at the initial solution selection stage and during self-improvement cycles. We explore four verifier strategies:

(1) *Random Selection:* As a baseline, we randomly select the frontier node from the pool of generated candidate solutions. We will always prioritize selecting from non-buggy candidates either in the initial solution selection phase or the self-improvement phase.

(2) *LLM-as-a-Judge:* We prompt LLM to assess the quality of each candidate solution by generating a numerical score given the task description and generated solution. Then each time, we will choose the solution with the highest score to debug or self-improve. The final solution will be the non-buggy one with the highest score.

(3) *Rubric-Based Grading (Rubric-as-a-Judge):* A structured grading rubric is first generated by GPT-4o based on the ground truth program, outlining multiple fine-grained solution milestones with associated scores. This rubric is then reviewed and refined by human expert to ensure accuracy and clarity (Chen et al., 2025). We input the task description, rubric and solution to LLMs to generate a numerical score for the solution. Then each time, we will choose the solution with the highest score to debug or self-improve. The final solution will be the non-buggy one with the highest score. This approach leverages the ground truth information indirectly as the feedback to help choose the most promising frontier solution at each step.

(4) *Elo Rating based Ranking (Elo, 1978; Zhou et al., 2025) and Selection:* We adapt the Elo Rating algorithm in Appendix A to maintain a dynamic ranking of candidate solutions. Each solution competes against others in pairwise comparisons using the LLM, and the ranking is updated accordingly. This enables the agent to select the best-performing node both among initial solutions and during recursive self-improvement. Importantly, it supports adaptive backtracking, allowing the agent to revisit previously under-ranked trajectories. An illustration of Elo Rating verifier is shown in Figure 2.

Initially, the system compares pairs of candidate branches, favoring the top-2 highest Elo-score solutions and start to explore the first best branch. If the solution is buggy, then first debug it until it's executable. Then the self-improvement phase starts. During the iterative refinement process, for each-step self-improvement in the best branch, we compare the new solution with the solution in

the second-best branch and select the highest-Elo-ranked solution for further refinement. Finally, the highest-score non-buggy solution is selected based on Elo ranking as the best solution.

Crucially, to enable adaptive and resilient reasoning, if a current path underperforms, the agent can backtrack and reconsider previously lower-ranked alternatives. This mechanism prevents premature commitment to suboptimal solutions and allows recovery from early mistakes.

In summary, by combining these above components, C&C achieves robust and flexible trajectory exploration and supports high-quality solution selection across a wide range of scientific tasks.

3 Experimental Setup

Baselines. To evaluate the effectiveness of C&C, we compare it against the following three baselines, each representing a different strategy for program generation and reasoning: 1) Direct Prompting: This baseline uses an LLM to generate a solution in a single pass using a task-specific prompt. It reflects the standard zero-shot setting commonly used in prior work. This method serves as a simple and widely adopted baseline for evaluating initial model generation quality. 2) Self-Debug (Chen et al., 2024): In this baseline, the model generates an initial solution and then attempts to improve it via self-debugging based on the Python interpreter execution feedback. While this method allows for limited reflection, it does not incorporate trajectory search, external verification, or ranking. It evaluates the isolated effect of a self-correction loop without broader solution exploration. 3) OpenHands (Wang et al., 2024): OpenHands is a general agent that is designed for multiple domains including Web and software engineering tasks. It builds on the ReAct framework (Yao et al., 2023) to generate the next action based on the previous observation. Instead of directly generating the entire program solution at once, OpenHands gradually finishes the solution step by step.

Dataset. To evaluate C&C in realistic scientific coding scenarios, we use ScienceAgentBench (Chen et al., 2025), a curated benchmark designed to assess agents' capabilities in scientific discovery. This dataset includes a diverse set of tasks that cover the entire workflow such as model development, data analysis, and visualization, spanning from four scientific disciplines: Bioinformatics, Computational Chemistry, Geographical Information Science, and Psychology & Cognitive Neuroscience. All of our experiments are done on their "without expert-provided knowledge" setting.

Experiment Details. We experiment with GPT-4o (both the 0513 version and the 1120 version) (OpenAI, 2024) and Claude-3.7-Sonnet (Anthropic, 2025). Since GPT-4o (2024-11-20) version is much cheaper than GPT-4o (2024-05-13) version, we try both versions for the main experiments, but use GPT-4o (2024-11-20) version for the ablation study and verifier mechanism part. For all experiments, we use the same hyperparameters. For temperature, we use 0.5 for code generation and 0.5 for debug, analysis and summary, and 0 when leveraging LLMs to compare or judge the solutions. For top-p, we use top 0.95, and perform 0-shot prompting via the APIs. For each baseline, we run three times to get the mean and standard deviation of the performance. For each verifier mechanism, we run the agent twice to get the mean and standard deviation of the performance.

To constrain the budget, we set the initial solution number to 5, maximum debug step to 3 and total exploration step to 10, which will include the self-improvement step if the budget has not been exhausted by self-debug.

Evaluation Metrics. We follow previous work (Chen et al., 2025) to comprehensively evaluate each generated program using three key metrics. (1) Valid Execution Rate (VER) measures whether a program can execute without errors. (2) Success Rate (SR) assesses whether the output satisfies the specific task goal, such as passing predefined criteria, matching expected predictions, or producing a high-quality visualization. These criteria are implemented as task-specific evaluation programs during the benchmark annotation process. SR is conditioned on VER: if a program fails to execute or save its output correctly, its SR is 0. (3) API Cost (Cost) reports the average dollar cost required to complete a single task using the agent. This metric accounts for API usage and serves to highlight the importance of designing cost-efficient agents, as emphasized by Kapoor et al., 2024.

4 Result Analysis

4.1 Main Results

Table 1 compares the performance of different agent strategies using two versions of GPT-4o (2024-05-13 and 2024-11-20) across three metrics: Success Rate (SR), Valid Execution Rate (VER),

Base Model	Agents	SR	VER	Cost ↓
GPT-4o (2024-05-13)	Direct Prompting	7.50 (0.5)	42.2 (1.6)	0.011 (0.000)
	OpenHands	13.1 (2.6)	62.8 (2.9)	1.093 (0.071)
	Self-Debug	14.7 (3.2)	71.2 (1.2)	0.057 (0.001)
	C&C (Ours)	**16.1** (1.2)	66.0 (3.5)	0.512 (0.009)
GPT-4o (2024-11-20)	Self-Debug	15.0 (4.8)	67.0 (7.4)	0.030 (0.010)
	C&C (Ours)	**18.6** (3.3)	**69.9** (0.6)	0.342 (0.008)

Table 1: Mean performances of each agent and standard deviations on ScienceAgentBench (Chen et al., 2025).

Models	Verification	SR	VER	Cost ↓
GPT-4o (2024-11-20)	Random Selection	15.2 (0.7)	64.7 (1.4)	0.180 (0.006)
	LLM-as-a-Judge	16.2 (3.5)	69.1 (0.7)	0.253 (0.013)
	Elo Ranking	**18.6** (3.3)	**69.9** (0.6)	0.342 (0.008)
GPT-4o (2024-11-20)	Rubric-as-a-Judge (w/ GT)	21.1 (2.1)	74.5 (2.8)	0.303 (0.016)

Table 2: Verification choice effect on our agent. “w/ GT" means using ground truth program judge signal.

and Cost. The experiment results show that our proposed agent C&C consistently outperforms baseline approaches across both model versions. Under the 2024-05-13 model, C&C achieves an SR of 16.1%, surpassing Self-Debug (14.7%) and OpenHands (13.1%), while maintaining a strong VER of 66.0%. Although the cost (0.512) is higher than Self-Debug (0.057) and Direct Prompting (0.011), C&C offers a better balance between performance and cost-effectiveness compared to OpenHands (1.093), which is substantially more expensive despite lower SR and VER.

With the updated GPT-4o (2024-11-20), in order to save the cost, we only choose the best baseline Self-Debug for comparison. C&C achieves a large performance gain compared to Self-Debug, with a 24% improvement for SR and a 2.9-point improvement for VER. Besides, the performance of C&C is more stable than Self-Debug, with a 31% reduction of standard deviation for SR and a 6.8-point reduction of standard deviation for VER.

4.2 Verifier Strategy Analysis

Table 2 evaluates the impact of different verification strategies on agent performance using GPT-4o (2024-11-20). Among the random selection, LLM-as-a-Judge and Elo Ranking, Elo Ranking achieves the highest SR (18.6%) and VER (69.9%), outperforming both other two baselines, which attain lower SR of 15.2% and 16.2%, respectively and lower VER of 64.7% and 69.1% respectively.

The Rubric-as-a-Judge achieves the highest SR and VER among all verification methods, benefiting from access to ground truth rubric descriptions that detail the correct solution steps. While this method partially leverages ground truth signals, it highlights the potential performance gains from incorporating reliable supervision when available. However, such rubric-based evaluations are often impractical in real-world scientific tasks, where detailed grading criteria are rarely accessible. In contrast, our Elo Rating-based verifier offers a more generalizable solution, as it operates independently of ground truth labels while still delivering strong performance. Due to its effectiveness and applicability in reality, we adopt Elo Ranking as the default verification method in our main agent framework.

4.3 Component Ablation Study

Table 3 presents a component-wise ablation study evaluating the impact of the number of initial solutions and the use of the self-improvement mechanism in our agent framework. We obvserve that:
(1) Number of successful initial solutions directly contributes to end-to-end success rate. The table shows a strong correlation between the average number of successful initial solutions and the overll success rate. Without self-improvement, increasing the number of initial solutions from 1 to 5 yields a steady improvement in the average number of successful initial solutions (from 0.24 to 0.98), the average number of successful nodes (from 0.40 to

Num of Init. Solutions	Use Self-Improve?	Avg # of Successful Init. Solutions	Avg # of Successful Nodes	Success Rate
1	No	0.24	0.40	40.5
2	No	0.40	0.57	40.5
5	No	0.98	1.14	45.2
5	Yes	**1.17**	**2.69**	**57.1**

Table 3: Component ablation on our agent. Model: GPT-4o (2024-11-20). The experiment is done on 42 selected tasks, where each task has been solved at least once by either a baseline or our agent.

1.14) and the overall success rate (from 40.5% to 45.2%). This indicates that generating multiple initial solutions increases the chance that at least one initial solution is close to correct, thus improving the agent's final performance. The more successful starting points the agent has, the more likely it is to select or build upon a valid reasoning path.

(2) Self-improvement enables the agent to generate more correct programs and achieve the highest success rate. The final row of Table 3 isolates the effect of enabling self-improvement: for the same initial solution size 5, when self-improvement is disabled (row 3), the agent achieves an average of 1.14 successful nodes and 45.2% success rate, while when self-improvement is enabled (row 4), the number of successful nodes jumps to 2.69, and the success rate increases significantly to 57.1%.

This demonstrates that self-improvement nearly doubles the number of correct programs, allowing the agent to refine and expand upon flawed or incomplete initial solutions. The improvement in both node-level correctness and overall task success confirms that self-improvement is a key driver of end-to-end performance.

Overall, the findings highlight the complementary roles of solution diversity and iterative refinement in enhancing agent performance on the tasks.

4.4 Error Analysis

To evaluate the effectiveness of our verifier-guided approach, we conducted an error analysis on 20 randomly selected unsuccessful tasks from ScienceAgentBench using C&C. We categorize errors into two main types: exploration errors and verification errors. An exploration error occurs when the agent's entire trajectory fails to produce any solution that meets the success criteria. In contrast, a verification error arises when at least one successful solution exists in the trajectory, but the agent fails to identify or select it as the final output.

We further subdivide exploration errors into two

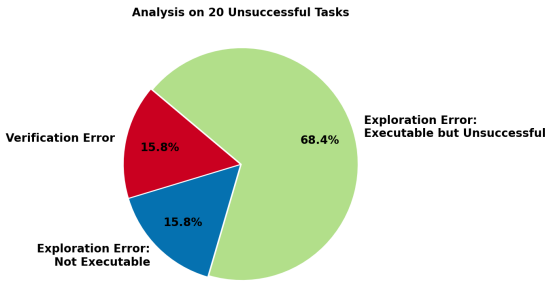

Figure 3: Error analysis on 20 randomly selected unsuccessful tasks of ScienceAgentBench.

categories: not executable, where all generated solutions are buggy, and executable but unsuccessful, where some solutions are runnable but do not satisfy the task requirements. As shown in Figure 3, only 15.8% of the failures were due to verification errors, suggesting that our Elo Rating-based verifier is generally effective at recognizing correct solutions. The remaining 84.2% of failures were attributed to exploration errors, with 15.8% resulting from non-executable programs and 68.4% from executable but incorrect outputs. This analysis highlights that the dominant source of failure lies in the exploration stage. It suggests that more test-time compute is needed in order to cover successful solution in the trajectory, such as increasing the initial solution size or self-improve more steps.

4.5 Initial Solution and Verifier Model Selection

Table 4 presents an ablation study on how different configurations of initial solution generators and verifiers affect agent performance. Using GPT-4o for both components yields a strong baseline (SR = 18.6%, VER = 69.9%) with the lowest agent cost (0.342). Interestingly, during our experiments, we observe that the successful solutions generated by Claude-3.7 and GPT-4o partially differ, with each model solving some examples that the other does not. This complementary behavior motivates us to

Models		SR	VER	Cost ↓		
Initial Solutions	Verification			Initial Solutions	Verification	Agent
GPT-4o	GPT-4o	18.6	69.9	**0.061**	**0.150**	**0.342**
mixture of GPT-4o and Claude-3.7	GPT-4o	**19.6**	63.7	0.108	0.164	0.417
Claude-3.7	GPT-4o	18.6	**72.5**	0.162	0.159	0.463
Claude-3.7	Claude-3.7	12.7	63.7	0.178	0.260	0.893

Table 4: Model selection of initial solutions and verification effect on our agent.

ensemble both models to enhance the diversity and robustness of the initial solution generator. Table 4 shows that replacing the initial solutions with a mixture of GPT-4o and Claude-3.7 leads to the highest SR (19.6%), albeit with a drop in VER (63.7%) and a moderate increase in cost (0.417).

Using Claude-3.7 exclusively as generator while retaining GPT-4o as the verifier maintains a competitive SR (18.6%) and achieves the highest VER (72.5%), but increases total cost to 0.463. However, when Claude-3.7 is used as both generator and verifier, performance deteriorates significantly (SR = 12.7%, VER = 63.7%) and cost nearly doubles (0.893). We suspect this is because GPT-4o has stronger verification capabilities than Claude-3.7.

5 Related Work

LLMs for Science. The integration of LLMs into scientific research workflows has opened new avenues for automating complex tasks, from hypothesis generation to data analysis. Recent benchmarks such as ScienceAgentBench (Chen et al., 2025) and DiscoveryBench (Majumder et al.) have been instrumental in evaluating LLMs' capabilities in data-driven scientific discovery. At the same time, different agents are applied or designed to solve these tasks. For example, OpenHands (Wang et al., 2024) completes the coding tasks by generating and executing partial code iteratively, using intermediate observations to inform subsequent actions. Self-Debug (Chen et al., 2024) corrects the code solution by providing the execution logs and feedback to LLMs. AIDE (Jiang et al., 2025) attempts to incorporate tree search to iteratively refine the code. However, these agents either rely on a single refinement strategy (Wang et al., 2024; Chen et al., 2024), or depend on gold verification signals such as explicit success metrics to determine whether a solution meets the task criteria and to guide further refinement (Jiang et al., 2025).

Test-time Scaling in LLMs. Prior work such as PlanSearch (Wang et al., 2025) and CodeMonkeys (Ehrlich et al., 2025) demonstrates that increasing the number of generated candidate solutions leads to an approximately log-linear improvement in the proportion of problems successfully solved by at least one candidate. This test-time compute scaling phenomenon significantly enhances overall solution coverage, that can be measured by metrics such as Pass@K in code generation tasks. SFS (Light et al., 2025) reveals performance gains on programming tasks by enhancing the solution diversity and leveraging prior search experiences. Snell et al., 2025 shows that scaling LLM test-time compute optimally can be more effective than scaling model parameters. Inspired by these work, we leverage the test-time scaling across multiple components of our agent. In addition, we uniquely leverage Elo Rating-based verifier to compare and refine candidate solutions during tree search. This enables effective exploration and improvement without requiring explicit success criteria, making our approach more generalizable to real-world scientific tasks where such gold signals are often unavailable.

6 Conclusion

In summary, we propose C&C, a verifier-guided agent framework designed to address the unique challenges of scientific coding tasks: uncertain solution spaces and the absence of clear-cut task success signals. By applying test-time compute scaling in various components, C&C generates diverse candidate solutions and iteratively refines them through self-debug and self-improvement, guided by an Elo Rating-based verifier. This framework enables the agent to expand and correct its trajectory without requiring gold labels or predefined metrics as the task success signal. Our experiments on ScienceAgentBench demonstrate that C&C consistently outperforms prior baselines, including direct prompting, OpenHands, and Self-Debug, as well as alternative verification strategies. These results demonstrate the effectiveness of our agent design and highlight the importance of generalizable verifiers in domains lacking gold verification signals.

Limitations

One limitation of our current framework is that it requires generating an entire code block in a single step, without support for incremental synthesis or partial execution feedback. As a result, for tasks that demand generating very long programs which are beyond the model's capacity, our agent may become unsuitable for effectively handling such cases. Additionally, once the agent enters the self-improvement stage, it lacks the ability to autonomously determine when to stop; instead, it continues until the predefined maximum number of total steps is reached. This rigid stopping criterion can result in unnecessary computation or missing a high-quality solution that has already been found earlier.

References

Mohamad Ali-Dib and Kristen Menou. 2024. Physics simulation capabilities of llms. *Preprint*, arXiv:2312.02091.

Anthropic. 2025. Claude 3.7 sonnet.

Debjyoti Bhattacharya, Harrison J Cassady, Michael A Hickner, and Wesley F Reinhart. 2024. Large language models as molecular design engines. *Journal of Chemical Information and Modeling*, 64(18):7086–7096.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. 2025. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. 2025. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. In *The Thirteenth International Conference on Learning Representations*.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. Codemonkeys: Scaling test-time compute for software engineering. *Preprint*, arXiv:2501.14723.

Arpad E. Elo. 1978. *The Rating of Chessplayers, Past and Present*. Arco Pub., New York.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Chenxing Wei, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Xiangru Tang, and 8 others. 2024. Data interpreter: An llm agent for data science. *Preprint*, arXiv:2402.18679.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. 2025. Aide: Ai-driven exploration in the space of code. *Preprint*, arXiv:2502.13138.

Qiao Jin, Zhizheng Wang, Yifan Yang, Qingqing Zhu, Donald Wright, Thomas Huang, W John Wilbur, Zhe He, Andrew Taylor, Qingyu Chen, and Zhiyong Lu. 2024. Agentmd: Empowering language agents for risk prediction with large-scale clinical tool learning. *Preprint*, arXiv:2402.13225.

Sayash Kapoor, Benedikt Stroebl, Zachary S Siegel, Nitya Nadgir, and Arvind Narayanan. 2024. Ai agents that matter. *arXiv preprint arXiv:2407.01502*.

Jonathan Light, Yue Wu, Yiyou Sun, Wenchao Yu, Yanchi Liu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, and Wei Cheng. 2025. SFS: Smarter code space search improves LLM inference scaling. In *The Thirteenth International Conference on Learning Representations*.

Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Bhavana Dalvi Mishra, Abhijeetsingh Meena, Aryan Prakhar, Tirth Vora, Tushar Khot, Ashish Sabharwal, and Peter Clark. Discoverybench: Towards data-driven discovery with large language models. In *The Thirteenth International Conference on Learning Representations*.

Ludovico Mitchener, Jon M Laurent, Benjamin Tenmann, Siddharth Narayanan, Geemi P Wellawatte, Andrew White, Lorenzo Sani, and Samuel G Rodriques. 2025. Bixbench: a comprehensive benchmark for llm-based agents in computational biology. *arXiv preprint arXiv:2503.00096*.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, Dieuwke Hupkes, Ricardo Silveira Cabral, Tatiana Shavrina, Jakob Foerster, Yoram Bachrach, William Yang Wang, and Roberta Raileanu. 2025. Mlgym: A new framework and benchmark for advancing ai research agents. *Preprint*, arXiv:2502.14499.

OpenAI. 2024. Gpt-4o.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*.

Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark Gerstein. 2024. Biocoder: A

benchmark for bioinformatics code generation with large language models. *Preprint*, arXiv:2308.16458.

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.

Evan Z Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean M. Hendryx, Summer Yue, and Hugh Zhang. 2025. Planning in natural language improves LLM search for code generation. In *The Thirteenth International Conference on Learning Representations*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Botao Yu, Frazier N. Baker, Ziqi Chen, Xia Ning, and Huan Sun. 2024. LlaSMol: Advancing large language models for chemistry with a large-scale, comprehensive, high-quality instruction tuning dataset. In *First Conference on Language Modeling*.

Botao Yu, Frazier N. Baker, Ziru Chen, Garrett Herb, Boyu Gou, Daniel Adu-Ampratwum, Xia Ning, and Huan Sun. 2025. Tooling or not tooling? the impact of tools on language agents for chemistry problem solving. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 7620–7640, Albuquerque, New Mexico. Association for Computational Linguistics.

Changzhi Zhou, Xinyu Zhang, Dandan Song, Xiancai Chen, Wanli Gu, Huipeng Ma, Yuhang Tian, Mengdi Zhang, and Linmei Hu. 2025. Refinecoder: Iterative improving of large language models via adaptive critique refinement for code generation. *Preprint*, arXiv:2502.09183.

A Appendix

A Elo Algorithm

Firstly, we initialize the Elo score $E_i = 1500$ for each code response P_i. Then, we iteratively update the Elo scores by using the relative scores between any two responses. Taking the response pair P_i and P_j as an example, we obtain their match results S_i and S_j:

Expected scores: When response P_i faces response P_j, the expected score for P_i (denoted E_i) is:

$$E_i = \frac{1}{1 + 10^{(R_j - R_i)/400}} \quad (1)$$

Similarly, the expected score for P_j is $E_j = 1 - E_i$.

After the match:

- If P_i wins: $S_i = 1$, $S_j = 0$
- If P_i loses: $S_i = 0$, $S_j = 1$
- If it's a draw: $S_i = 0.5$, $S_j = 0.5$

Rating update rule: Use the standard Elo update formula. For response P_i:

$$R_i' = R_i + K \cdot (S_i - E_i) \quad (2)$$

Where:

- R_i is the old rating, R_i' is the new rating.
- S_i is the actual score.
- E_i is the expected score.
- K is a constant. We set it to 32 to determine how fast ratings change.

For each pair of responses, we update both of their Elo scores once. After all pairwise comparisons, we obtain the final Elo scores for all responses, which can be used to derive a ranking.