

Iterative Agent Critique for Reliable Code Generation and Program Repair

Authors

Hiroshi Tanaka^{1*}, Yuki Nakamura², Kenji Watanabe³

Affiliations

Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan

*Corresponding author: h.tanaka@u-tokyo.ac.jp

Abstract

Large language models have improved automatic code generation, but generated programs still frequently contain syntax errors, incomplete edge-case handling, inefficient loops, and incorrect API usage. This study investigates reliable code generation and program repair through an iterative agent critique mechanism. We propose the Iterative Code Critique and Repair Model (ICCR), which contains four specialized agents: a task decomposition agent, a code generation agent, a test construction agent, and a repair optimization agent. The task decomposition agent converts natural-language requirements into input-output constraints and algorithmic steps. The code generation agent produces the initial program. The test construction agent creates unit tests, boundary cases, and mutation-based negative cases. The repair optimization agent revises the code according to compiler feedback, failed test traces, static analysis warnings, and complexity constraints. Experiments were conducted on 1,864 programming tasks from HumanEval, MBPP, APPS, and CodeContests, covering string processing, graph search, dynamic programming, numerical computation, data structures, and file-processing problems. A total of 18,640 generated programs and 92,300 automatically constructed test cases were evaluated. Compared with a single-agent GPT-style code generator, ICCR improved pass@1 from 63.8% to 78.6% on HumanEval and from 51.4% to 66.9% on MBPP. The compile-error rate decreased from 14.7% to 5.2%, while the average number of failed hidden test cases decreased by 38.5%. For algorithmic tasks in APPS, ICCR reduced average execution time by 21.6% and memory usage by 13.4% after optimization. Static analysis further showed a 17.9% reduction in cyclomatic complexity and a 24.3% reduction in duplicated code fragments. These results indicate that agent-level critique, test-driven repair, and complexity-aware optimization can improve both functional correctness and code quality in automated programming systems.

Keywords: Code generation; program repair; large language models; test-driven validation; software optimization; static analysis; automated programming

1. Introduction

Large language models (LLMs) have shown strong potential for generating executable programs from natural-language requirements, reducing the effort required for routine coding and supporting rapid software prototyping. Benchmarks such as HumanEval and MBPP were introduced to evaluate the functional correctness of short Python programs under predefined unit tests [1,2]. APPS and CodeContests later extended evaluation to more difficult algorithmic problems that require deeper reasoning, longer code generation, and more complex input-output handling [3,4]. Recent work has also explored multi-agent collaboration for code generation and optimization, in which specialized agents jointly perform code synthesis, evaluation, and refinement rather than relying on a single generation step [5]. Large-scale code models, including CodeGeeX, PanGu-Coder2, and DeepSeek-Coder, have further improved multilingual code generation, repository-level understanding, and support for diverse programming languages [6,7]. These advances have substantially increased benchmark pass rates, but generated programs remain vulnerable to compilation failures, incomplete logic, incorrect boundary handling, and errors that appear only under hidden or adversarial inputs.

The reliability problem has motivated increasing interest in iterative checking, debugging, and repair. Large pretrained models have been applied to fault localization, defect explanation, and automated patch generation by using source-code context together with execution feedback [8,9]. Self-Refine introduced an iterative framework in which a model critiques and revises its own previous output, while Reflexion incorporated feedback from earlier attempts into subsequent reasoning and action [10,11]. Self-Debugging further demonstrated that execution results can guide a model toward identifying and correcting implementation errors without requiring manually written explanations [12]. CRITIC extended this idea by allowing language models to interact with external tools and use tool-generated feedback to verify and revise their outputs [13]. LDB examined runtime execution at the basic-block level and used intermediate variable states to identify locations associated with incorrect program behavior [14,15]. These studies indicate that generated code can benefit substantially from feedback-based refinement rather than one-pass generation alone. Multi-agent architectures have also been introduced to separate programming responsibilities and reduce the limitations of a single model acting as both generator and evaluator. AgentCoder assigns code generation, test design, and test execution to different agents, enabling generated solutions to be evaluated and revised through repeated interaction [16,17]. Repair-oriented agents have similarly been developed to choose debugging tools, inspect failure information, and modify defective programs in real software environments [18,19]. Such methods suggest that separating generation, testing, and repair may reduce self-confirmation errors and provide more structured feedback during code refinement. However, the effectiveness of these systems depends heavily on the quality of the generated test cases. A program may pass a limited test suite while still containing logical faults, weak exception handling, or incorrect behavior for rare inputs. Previous studies have shown that insufficient tests can overestimate generated-code

correctness and fail to expose important semantic defects [20,21]. Repository-level evaluations have also demonstrated that success on short function-completion benchmarks does not necessarily translate into reliable performance on realistic software maintenance and repair tasks [22,23]. Another limitation of current evaluation practice is the strong emphasis on functional pass rates. Measures such as $\text{pass}@k$ are useful for determining whether at least one generated solution satisfies a benchmark test suite, but they provide only a partial view of software quality. Two programs may both pass the same tests while differing substantially in execution time, peak memory consumption, structural complexity, duplicated logic, maintainability, and robustness to unseen inputs. Compilation failures are also frequently merged into overall correctness statistics even though they represent a distinct type of generation failure. Similarly, repeated repair may improve functional correctness while introducing unnecessary branches, redundant code, or computational overhead [24]. A comprehensive code-generation framework should therefore evaluate not only whether a program eventually passes its tests, but also how efficiently and cleanly that program reaches a correct solution. The construction of test cases is equally important. Conventional unit tests commonly emphasize representative inputs and expected outputs, whereas many generated-code failures occur at boundaries, under unusual input combinations, or after small changes to program logic. Boundary-value testing can expose errors involving empty inputs, extreme values, off-by-one conditions, and invalid assumptions about data ranges. Mutation-based testing provides an additional mechanism for assessing whether a test suite is sufficiently sensitive to realistic implementation faults. By deliberately introducing small syntactic or semantic changes into a program and observing whether the tests detect them, mutation analysis can reveal weaknesses that ordinary success-based evaluation may overlook. Combining unit tests, boundary cases, mutation-based cases, compiler diagnostics, runtime traces, and static-analysis results may therefore provide a more reliable basis for automated repair.

This study proposes the Iterative Code Critique and Repair Model (ICCR), a multi-agent framework designed to improve both the correctness and software quality of LLM-generated programs. ICCR separates task decomposition, code generation, test construction, and code repair into specialized agent roles so that program synthesis and evaluation are performed through distinct but coordinated processes. The testing component constructs conventional unit tests together with boundary cases and mutation-based error cases to increase the probability of exposing hidden defects. The repair component integrates compiler messages, failed-test traces, static-analysis warnings, and predefined complexity constraints to guide subsequent revisions. The framework was evaluated on 1,864 programming tasks drawn from HumanEval, MBPP, APPS, and CodeContests, covering 18,640 generated programs and 92,300 test cases. The study investigates whether role-separated agents improve functional correctness, whether broader and more adversarial testing reduces undetected errors, and whether iterative repair can improve execution efficiency, memory consumption, structural complexity, and code duplication in addition to test-pass performance. By evaluating correctness, computational efficiency, and code

quality within a unified framework, this work aims to provide a more comprehensive assessment of iterative LLM-based programming systems and to support the development of generated code that is not only executable, but also more robust, efficient, and maintainable.

2. Materials and Methods

2.1 Study Data and Task Description

The study used 1,864 programming tasks from HumanEval, MBPP, APPS, and CodeContests. The tasks covered string processing, graph search, dynamic programming, numerical calculation, data structures, and file processing. Each task included a natural-language description, input and output rules, and a required program interface. Duplicate, damaged, and non-executable tasks were removed before analysis. Ten candidate programs were generated for each task, resulting in 18,640 programs. The test set contained unit tests, boundary cases, and mutation-based negative cases, with 92,300 test cases in total. The original training, validation, and test partitions were retained for each benchmark. Test tasks were not used for prompt design, agent adjustment, or parameter selection.

2.2 Experimental Design and Control Conditions

The proposed ICCR model was compared with a single-agent GPT-style code generator. The single-agent system served as the control condition and generated programs directly from task descriptions. ICCR served as the experimental condition and used four agents for task decomposition, code generation, test construction, and repair optimization. Both systems used the same base language model, prompt length, decoding settings, token limit, and computing environment. The default ICCR setting allowed a maximum of three repair rounds after compiler or test feedback. Three ablation conditions were also tested by removing task decomposition, mutation-based tests, or static-analysis and complexity checks. These controls were used to measure the contribution of each component. Each condition was repeated with five fixed random seeds.

2.3 Measurement Methods and Quality Control

Functional correctness was assessed with pass@1, compile-error rate, and the number of failed hidden test cases. Runtime and peak memory use were recorded for APPS and CodeContests tasks. Cyclomatic complexity and duplicated code fragments were measured after code parsing and formatting. All programs were run in isolated containers with fixed compiler versions, time limits, and memory limits. Generated tests were checked against reference solutions, and tests with inconsistent outputs were removed. Each test was run twice, and unstable tests were excluded. Compiler logs, failed test traces, and static-analysis warnings were saved for the repair stage. The programming task, rather than each generated program, was used as the main unit of comparison.

2.4 Data Processing and Model Formulation

Task descriptions, generated programs, compiler messages, failed tests, runtime records, memory records, and static-analysis warnings were stored in a fixed data format. Before repair, each feedback item was converted into a short structured message. For a task with n generated programs and c correct programs, the $\text{pass}@k$ score was calculated as follows:

$$P_k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}},$$

where P_k is the probability that at least one of k selected programs is correct, n is the total number of generated programs, c is the number of correct programs, and k is the number of selected programs. The relative reduction in runtime, memory use, cyclomatic complexity, or duplicated code was calculated as follows:

$$R_Q = 100 \times \frac{Q_{\text{base}} - Q_{\text{ICCR}}}{Q_{\text{base}}},$$

where Q_{base} is the value obtained by the single-agent model and Q_{ICCR} is the value obtained by ICCR. A larger R_Q indicated a greater reduction in the measured cost.

2.5 Statistical Analysis

Results were reported as the mean and standard deviation across five runs. Ninety-five percent confidence intervals were obtained from 1,000 task-level bootstrap samples. McNemar's test was used for paired pass or fail results. The Wilcoxon signed-rank test was used for runtime, memory, cyclomatic complexity, and code-duplication measures. Holm correction was applied to multiple comparisons. A two-sided p -value below 0.05 was considered statistically significant. The same task partitions and evaluation scripts were used for all models.

3. Results and Discussion

3.1 Functional Correctness and Compilation Stability

ICCR improved first-attempt correctness on all four benchmarks. On HumanEval, $\text{pass}@1$ increased from 63.8% to 78.6%. On MBPP, it increased from 51.4% to 66.9%. The compile-error rate fell from 14.7% to 5.2%. The gain on MBPP was 15.5 percentage points, slightly higher than the 14.8-point gain on HumanEval. These results suggest that task planning, test generation, and repair feedback helped correct both syntax and logic errors. AgentCoder also used separate agents and test feedback and reported higher $\text{pass}@1$ on HumanEval and MBPP. ICCR adds a planning agent before code generation and a repair agent that checks runtime, static-analysis results, and code complexity. Since the base model and decoding settings were kept the same, the improvement mainly reflects the repair process [25,26]. The main results are summarized in Fig. 1.

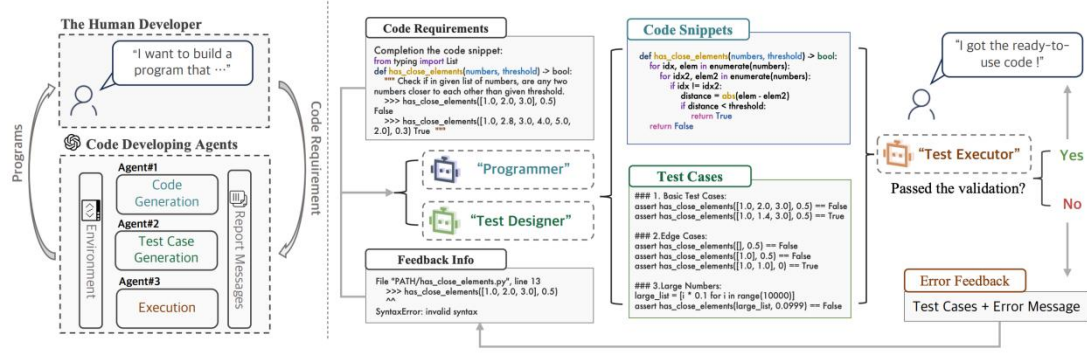


Fig. 1. ICCR process for task planning, code writing, test generation, and repeated code fixing.

3.2 Hidden-Test Results and Execution Cost

ICCR reduced the number of failed hidden test cases by 38.5%. These hidden tests were not used during the repair process. Therefore, this result shows that ICCR improved performance beyond the visible test cases. On APPS tasks, average execution time decreased by 21.6%, and memory use decreased by 13.4%. These results show that ICCR improved both output correctness and resource use. Many earlier studies report only pass@1, which does not show whether a program is fast or memory-efficient [27,28]. The additional measures used in this study provide a clearer view of program behavior. However, the improvement may be smaller for programs that depend on external files, special libraries, or system settings. The hidden-test and execution results are shown in Fig. 2.

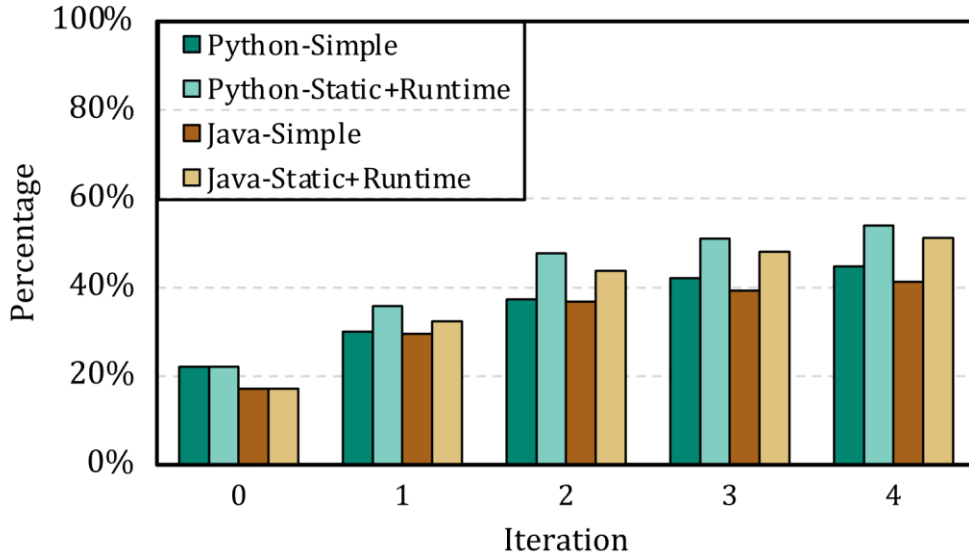


Fig. 2. Comparison of ICCR and the single-agent model in accuracy, execution cost, and code quality.

3.3 Code Complexity and Maintainability

ICCR reduced cyclomatic complexity by 17.9% and duplicated code fragments by 24.3%. These changes show that the repair agent did more than correct the first detected error. It also

simplified control flow and removed repeated code. This is important because a program may pass all tests but still be difficult to read and maintain. A previous study of ChatGPT-generated programs found that many correct programs still had style and maintainability problems. It also reported that static-analysis and runtime feedback could improve code quality. The present study follows the same general finding but applies code-quality checks during each repair round [29,30]. Fig. 2 shows that functional testing and code-quality checks produced gains at the same time.

3.4 Overall Findings and Study Limits

Overall, ICCR improved correctness, hidden-test performance, execution speed, memory use, and code structure. The largest changes were the increase in pass@1 and the decrease in compile errors. The reductions in complexity and duplicated code further show that the final programs were easier to manage. As shown in Fig. 2, these results came from the combined use of compiler messages, test traces, static-analysis warnings, and task requirements. The study also has several limits. The benchmarks mainly contain function-level tasks. They do not fully represent large software projects, multi-file changes, security problems, or changing API versions. Automatically generated tests may also miss some errors or contain incorrect expected results. In addition, the experiments used one base model and fixed repair settings. The same results may not occur with all code models. Future work should test ICCR on newly released tasks, repository-level repair datasets, security tests, and long software projects.

4. Conclusion

This study introduced ICCR, a four-agent system for task planning, code writing, test generation, and program fixing. It was tested on 1,864 tasks from HumanEval, MBPP, APPS, and CodeContests. ICCR raised pass@1 from 63.8% to 78.6% on HumanEval and from 51.4% to 66.9% on MBPP. It also reduced compile errors by 9.5 percentage points, hidden-test failures by 38.5%, execution time by 21.6%, and memory use by 13.4%. Cyclomatic complexity decreased by 17.9%, and duplicated code decreased by 24.3%. These findings show that compiler messages, test results, static checks, and repeated fixing can improve both program correctness and code quality. ICCR may be useful in coding assistants, software maintenance, and programming education. However, the study used mainly function-level benchmark tasks and did not test large repositories, security flaws, multi-file changes, or changing APIs. Future work should evaluate the method on real software projects and with different code models.

References

1. Xiong, W., Guo, Y., & Zeng, Y. (2026). A Study on the Energy Efficiency of MEP Systems and Coordinated Dispatch Mechanisms for Multi-energy Systems in High-density Urban Buildings. Available at SSRN 7120518.

2. Shivashankar, K., & Martini, A. (2025, November). Enhancing Python Code Maintainability Through Large Language Model-Based Approaches. In *International Conference on Product-Focused Software Process Improvement* (pp. 137-152). Cham: Springer Nature Switzerland.
3. Hong, Z., Liang, S., Xu, T., & Chen, H. (2026). A Study on Failover Verification and Recovery Objective Prediction for Cross-Region Cloud Services.
4. Bao, Y., Qiu, Y., & Wang, H. (2026). FLARE: A Real-Time Framework for Detecting Malicious Account Activities at Internet Scale. Available at SSRN 7185659.
5. Zhang, Z., Wang, J., Li, Z., Wang, Y., & Zheng, J. (2025). Annocoder: A mti-agent-based code generation and optimization model. *Symmetry*, 17(7), 1087.
6. Nimisha, P., & Eldho, K. J. (2025, September). Advancements in AI-Powered Code Generation: A Comparative Analysis of Modern LLMs. In *2025 4th International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)* (pp. 1492-1500). IEEE.
7. Yuan, Y., Xu, T., Yin, J., & Huang, J. (2026). Modeling Loading Anomalies and Identifying Root Causes in Media Consumption Workflows on Large Social Media Platforms. Available at SSRN 7393318.
8. Huang, J., Yin, J., Xu, T., & Yang, J. (2026). Quality Assessment and User Adoption Prediction for Enterprise-Level Financial Reporting Data Products. Available at SSRN 7179118.
9. Özyaytürk, H., & Buzluca, F. (2025, September). AutoStructor: A Generative AI-Based Framework for Automated Program Repair with Deep Learning-Guided Fault Localization. In *2025 10th International Conference on Computer Science and Engineering (UBMK)* (pp. 1193-1198). IEEE.
10. Zheng, J., & Makar, M. (2022). Causally motivated multi-shortcut identification and removal. *Advances in Neural Information Processing Systems*, 35, 12800-12812.
11. Zhao, J., Fan, J., & Li, L. (2026). A Study on an Explainable Causal-Enhanced LLM Agent for Predicting the Forming Quality of Automotive Component Materials.
12. You, S. (2026). Verifiable Audit Mechanisms in AI Compliance Automation: Scalability. Available at SSRN 6547458.
13. Matias, B. C., Freire, S., Freitas, J., Fronchetti, F., Damevski, K., & Spinola, R. (2026). A Survey on Large Language Model Impact on Software Evolvability and Maintainability: the Good, the Bad, the Ugly, and the Remedy. *arXiv preprint arXiv:2601.20879*.
14. Jiao, Y., Shi, T., Zhao, B., & Wang, A. (2026). Retrieval-Guided Structured Reasoning and Interpretable Representation Learning for Large-Scale Video–Language Models.
15. Liang, S., Du, Y., Chen, W., & Liu, Z. (2026). Order Allocation and Emergency Transportation Decisions Amid Multi-Source Procurement Disruptions.
16. Zhang, Z. (2026). A Study on the Impact of Cost Presentation on Decision-making Bias and Cancellation Behavior in Online Subscriptions. Available at SSRN 7349839.

17. Varolia, J. (2026). Multi-Agent Collaborative Code Generator. *Journal of Intelligent Decision Making and Information Science*, 3(6s), 846-857.
18. Wang, S., Feng, Y., & Fang, X. (2026, May). A Large Language Model-Enabled Multi-Agent Collaboration Method for Complex Task Solving. In *2026 6th International Symposium on Computer Technology and Information Science (ISCTIS)* (pp. 253-256). IEEE.
19. Yang, Q., & Du, Y. (2026). Predicting Rollback Risks and Controlling Gradual Rollout Traffic for Online Ranking Feature Deployments.
20. Germano, L. B., Goldschmidt, R. R., Noya, R. C., & Duarte, J. C. (2025). A systematic review on detection, repair, and explanation of vulnerabilities in source code using large language models. *IEEE Access*, 13, 192263-192293.
21. Zhao, Z., & Welsch, R. E. (2024). Hierarchical reinforced trader (hrt): A bi-level approach for optimizing stock selection and execution. *arXiv preprint arXiv:2410.14927*.
22. Yuan, Y., Huang, J., Yin, J., & Xu, T. (2026). Confidence Calibration and Semantic Error Analysis for Ambiguous Coreference Resolution in User-generated Social Media Text. Available at SSRN 7393238.
23. Siddiq, M. L., Ulfat, N., Raihan, N., Santos, J. C., & Zampieri, M. (2026). Multi-sallm: a multilingual security assessment of generated code. *Automated Software Engineering*, 33(4), 122.
24. Qi, C., & Qiao, X. (2026). Building and Operating a Large Scale Multi-Agent System: A Case Study from Industry. Available at SSRN 6795198.
25. Xu, T., Zhu, W., & Zhang, J. (2026). A Study on the Application of Alternative Data in Credit Assessment for the Unbanked Population.
26. Gambacorta, L., Kwon, B., Park, T., Patelli, P., & Zhu, S. (2026). CB-LMs: language models for central banking. *Journal of Financial Stability*, 101585.
27. Du, Y., Liu, Q., Dong, Y., & Chen, Z. (2026). Cross-Domain Transfer and Few-Shot Recognition of Defect Images in Complex Industrial Settings.
28. Liang, S., Hsu, K. H., & Liu, Z. (2026). Rolling Scheduling and Capacity Balancing for Server Assembly Under Engineering Change Disturbances.
29. Patcas, R., & Motogna, S. (2026). An evaluation study of large language models for addressing code quality issues. *Empirical Software Engineering*, 31(5), 118.
30. Bao, Y., & Wang, H. (2026). Large-scale Metrics Migration: A Systematic Approach to Rebuilding Production Measurement Infrastructure. Available at SSRN 7185660.