

APPLYING AI TO SUPPORT INSTRUCTORS IN PROGRAMMING ASSESSMENT: A PILOT STUDY

ỨNG DỤNG TRÍ TUỆ NHÂN TẠO HỖ TRỢ GIÁNG VIÊN TRONG ĐÁNH GIÁ LẬP TRÌNH: NGHIÊN CỨU THÍ ĐIỂM

Nguyen Ngoc Dan Thanh, Le Minh Tu

School of Engineering and Technology, Tra Vinh University

Received: 02/04/2026; Accepted: 09/04/2026; Published: 20/04/2026

Tóm tắt: Bài báo trình bày một nghiên cứu thí điểm về việc ứng dụng trí tuệ nhân tạo nhằm hỗ trợ giảng viên trong đánh giá mã nguồn. Trong các học phần lập trình nhập môn, việc đánh giá số lượng lớn bài nộp thường tốn nhiều thời gian và khó phát hiện các lỗi tiềm ẩn không thể hiện qua kiểm thử thông thường. Nghiên cứu đề xuất một phương pháp phân tích mã nguồn C dựa trên trí tuệ nhân tạo, cung cấp các chỉ báo về khả năng tồn tại lỗi thông qua xác suất và mức độ rủi ro. Hệ thống được tích hợp vào quy trình chấm bài nhằm hỗ trợ giảng viên trong việc rà soát và ưu tiên các bài nộp cần xem xét. Kết quả bước đầu cho thấy phương pháp hỗ trợ phát hiện các lỗi tiềm ẩn và cung cấp thêm thông tin trong quá trình đánh giá mã nguồn.

Từ khóa: trí tuệ nhân tạo; đánh giá lập trình; phân tích mã nguồn; hỗ trợ giảng viên; giáo dục lập trình.

Abstract: This paper presents a pilot study on the application of artificial intelligence to support instructors in source code assessment. In introductory programming courses, evaluating a large number of submissions is time-consuming and makes it difficult to identify hidden issues that are not revealed through standard test cases. To address this challenge, an AI-based approach is proposed to analyze C source code and provide indications of potential issues through probability estimates and risk levels. The system is integrated into the assessment workflow to assist instructors in reviewing and prioritizing submissions that require further inspection. Preliminary results suggest that the approach supports the detection of non-obvious issues and provides additional information to assist instructors during the code evaluation process.

Keywords: artificial intelligence; programming assessment; source code analysis; instructor support; programming education.

1. Introduction

In recent years, programming has become a fundamental component of many higher education curricula. Introductory programming courses often require instructors to evaluate a large number of student submissions within limited time constraints. This process is both time-consuming and cognitively demanding, as instructors must examine not only program outputs but also the underlying source code to ensure correctness and quality. Traditional assessment methods based on test cases and output comparison are often insufficient for detecting hidden issues such as improper memory handling or subtle logical errors.

To address these challenges, various automated assessment and feedback systems have been proposed. Prior studies have explored automated code evaluation and feedback models to improve efficiency and consistency [2], while systematic reviews highlight the importance of such systems in programming education [1]. More recent approaches incorporate artificial intelligence, including large language models, to enhance grading and feedback generation [3]. Additional tools, such as visualization-based systems and automated fault detection techniques, support program comprehension and debugging, although

challenges remain in reliability and generalization [4].

Alongside these developments, advances in artificial intelligence have improved the ability to analyze source code. Pre-trained models such as CodeBERT capture semantic relationships in programming languages [5]. Research in vulnerability detection and static analysis has further introduced deep learning-based approaches and traditional rule-based techniques for identifying potential issues in code [6].

Despite these advancements, In practice, instructors still rely heavily on manual code inspection to identify non-obvious issues. This gap highlights the need for tools that support instructors in reviewing student submissions more effectively.

To address this need, this paper presents a pilot study on the application of artificial intelligence to support instructors in analyzing student source code within the assessment workflow. The proposed system provides indications of potential issues to assist instructors in identifying submissions that require closer inspection, while maintaining human judgment as the basis for final evaluation decisions.

2. Related work

2.1. Programming Assessment and Feedback Systems

Automated assessment and feedback systems have been widely studied in programming education to improve the efficiency and consistency of evaluating student work. Early approaches primarily focused on test-case-based grading to assess program correctness, while more recent models incorporate additional criteria such as syntax checking, structural analysis, and feedback generation to support both evaluation and learning processes [2].

Systematic reviews highlight the importance of such systems in programming courses, particularly in large-class settings where individualized feedback is difficult to provide [1]. With the emergence of artificial intelligence, especially large language models, recent work has explored their use in automated grading and feedback generation, showing promising potential in enhancing feedback quality [3].

In addition to grading-oriented systems, tools have also been developed to support program comprehension and debugging. Visualization-based approaches help illustrate program execution and logic flow, while automated fault identification techniques provide support in detecting issues in student code, although challenges remain in accuracy and generalizability [4].

Despite these developments, most existing systems primarily focus on supporting learners. Comparatively less attention has been given to tools that directly assist instructors in examining student code and identifying potential issues during the assessment process.

2.2. AI Techniques for Code Analysis

Recent advances in artificial intelligence have significantly improved the ability to analyze source code. Pre-trained models such as CodeBERT can capture semantic relationships in programming languages and support tasks such as code classification and defect detection [5].

In parallel, research on software vulnerability detection and static analysis has produced a range of approaches for identifying potential issues in code. Traditional static analysis techniques remain relevant, incorporating rule-based, pattern-based, and machine learning-assisted methods [6].

These developments indicate that AI-based techniques can be effectively leveraged for analyzing student code. However, existing research has largely focused on model performance or general-purpose tools, with limited attention to educational applications. In particular, there remains a gap in applying AI-based code analysis to support instructors in assessing student submissions and identifying non-obvious issues beyond conventional evaluation methods.

3. Proposed system

3.1. System Architecture

The overall architecture consists of three main components: input processing, feature extraction, and issue prediction.

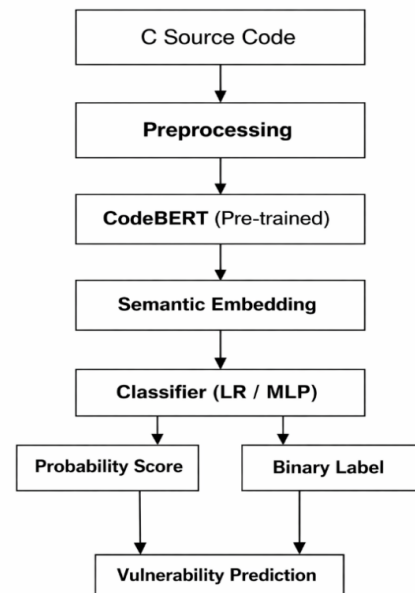


Figure 1. System architecture for analyzing student source code and identifying potential issues.

First, student submissions written in the C programming language are provided as input to the system. The source code is preprocessed for model input. A pre-trained code model generates fixed-length embeddings that capture syntactic and semantic information.

The extracted features are then fed into a classification component that estimates the likelihood of potential issues in the code. The system outputs a probability score, a predicted label, and a corresponding risk level, providing an initial indication of code quality and highlighting submissions for further inspection. To facilitate interpretation, three qualitative risk levels are defined, as summarized in Table 1.

Table 1. Definition of risk levels in the proposed system.

Risk Level	Description
LOW	No clear indication of risk is detected at the file level.
MEDIUM	Potential issues are indicated and require manual inspection.
HIGH	High risk is indicated, and further analysis and correction should be prioritized.

These risk levels are intended to provide instructors with an intuitive indication of code quality and to support prioritization during the assessment process.

3.2. User Interface and Functionalities

The interface is designed to be easily integrated into existing assessment workflows, focusing on delivering actionable indicators rather than detailed technical explanations.

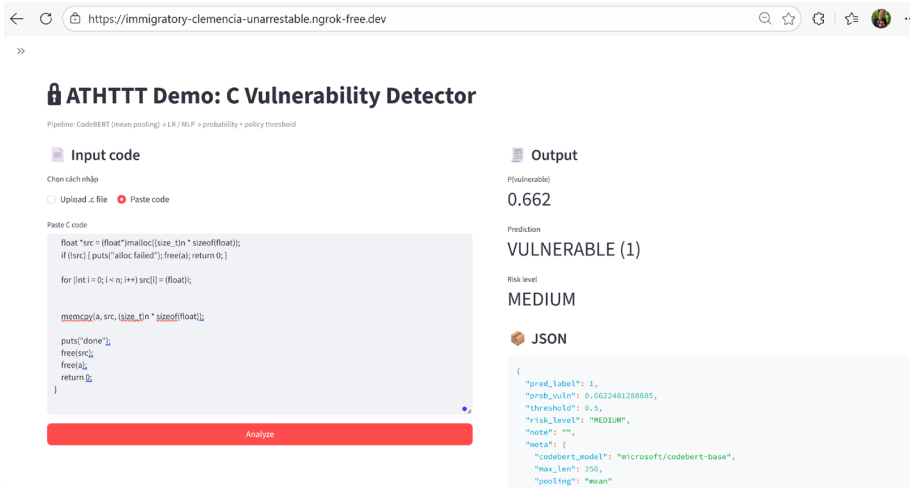


Figure 2. User interface of the proposed system for analyzing student source code.

As illustrated in Figure 2, the system analyzes a sample student program and provides a probability score along with a corresponding risk level. For example, a program involving dynamic memory allocation may be flagged as MEDIUM risk if potential issues such as improper memory handling are detected. This illustrates how the system supports instructors in identifying submissions that require further inspection.

3.3. Integration into Assessment Workflow

The proposed system functions as a supporting tool within the instructor's assessment process. In a typical workflow, students submit programming assignments through standard channels, and instructors use the system as an auxiliary step to review selected submissions.

By analyzing student code and highlighting potential issues, the system assists instructors in identifying cases that require further inspection, particularly non-obvious issues not captured by test cases, such as improper memory usage or subtle logical inconsistencies.

The system is not intended to replace the role of the instructor. Instead, it complements existing practices by supporting more informed review, while final evaluation and grading decisions remain under the responsibility of instructors.

4. Implementation in teaching practice

4.1. Context

The proposed system was implemented in an introductory programming course using the C language. The course is offered to first-year students, who complete assignments covering fundamental concepts such as control structures, arrays, and basic memory handling.

A set of student submissions from regular coursework was used for the pilot implementation. These submissions include both correct solutions

and code containing potential issues. The objective was to examine the feasibility of integrating the system into the instructor's assessment process.

4.2. Procedure

The implementation followed a two-step assessment process. First, student assignments were evaluated using conventional methods based on compilation

results and predefined test cases.

In the second step, the same submissions were analyzed using the proposed system, which generated a probability score, predicted label, and risk level. These outputs were used as supplementary information during instructor review.

Rather than replacing traditional assessment, the system provided an additional layer of analysis. Instructors focused on submissions flagged as higher risk, particularly those that passed test cases but might still contain underlying issues.

4.3. Evaluation Criteria

Given the exploratory nature of the study, evaluation focused on practical aspects rather than detailed statistical analysis. Three criteria were considered: (1) evaluation effort, based on instructor observation during the review process; (2) the system's ability to identify potential issues compared to conventional methods; and (3) the detection of non-obvious issues not revealed by standard test cases.

5. Results and discussion

5.1. Results

The pilot implementation provides initial observations on how the proposed system supports instructors in assessing student programming assignments. When integrated into the evaluation workflow, the system offers additional information beyond conventional test-case-based assessment.

Table 2 Performance comparison of models (pilot study).

Model	Accuracy	Precision	Recall	F1-score
Logistic Regression	0.613	0.591	0.733	0.655
MLP	0.573	0.543	0.920	0.683

The results in Table 2 show that the MLP model achieves higher recall, indicating a stronger ability to identify potentially problematic code. In contrast, Logistic Regression provides more balanced performance across metrics, suggesting more stable predictions with fewer false positives.

These findings highlight a trade-off between

detection coverage and prediction reliability. From an instructional perspective, higher recall helps identify more submissions requiring inspection, while a balanced model reduces unnecessary alerts. Model selection should therefore align with the needs of the assessment process.

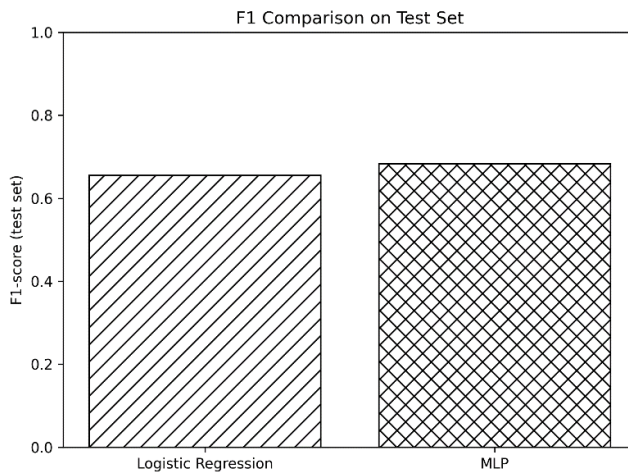


Figure 4. Comparison of F1-scores between Logistic Regression and MLP on the test set.

As shown in Figure 4, the MLP model achieves a slightly higher F1-score, indicating improved detection performance. However, this comes with an increased tendency for false positives, whereas Logistic Regression provides more balanced predictions.

The system is also able to highlight submissions that may contain issues despite producing correct outputs. In several cases, programs that passed predefined test cases were flagged as higher risk, prompting further inspection. This suggests that the system can assist in identifying non-obvious issues not detected by traditional methods.

In terms of efficiency, the system enables instructors to focus on submissions prioritized by risk levels, supporting more efficient review without replacing manual evaluation.

5.2. Discussion

The findings suggest that artificial intelligence can serve as a supporting tool in programming assessment by complementing traditional methods. By providing an additional layer of analysis beyond test-case evaluation, the approach helps identify issues that may otherwise be overlooked.

However, system outputs should be interpreted as indicators rather than definitive judgments, as false positives may occur, requiring instructor validation. As a pilot study, the findings should be considered an initial indication of feasibility rather than conclusive evidence.

In comparison with existing studies, the proposed approach differs in its focus. While prior work emphasizes automated grading and feedback for

learners [2], [3], this study focuses on supporting instructors during assessment. Unlike fault localization approaches [4], the system provides high-level indications of potential issues rather than precise error locations.

Overall, the results suggest that AI-based code analysis can support instructors in reviewing student submissions and identifying non-obvious issues, complementing existing assessment practices.

6. Conclusion and future work

This paper proposed approach integrates AI-based code analysis into the assessment workflow to provide indications of potential issues beyond conventional evaluation methods. The preliminary results suggest that the system can assist instructors in identifying non-obvious issues and prioritizing submissions for further inspection, while maintaining instructor judgment as the basis for final decisions. As a pilot study, the findings should be considered an initial indication of feasibility. Future work will focus on improving detection reliability, reducing false positives, and extending the approach to broader educational settings and additional programming languages.

References

- [1] Castellanos, H., Vieira, C., & Magana, A. J. (2026). Feedback Systems on Computer Programming Courses: A Systematic Literature Review. *IEEE Access*.
- [2] Parvathy, R., Thushara, M. G., & Kannimoola, J. M. (2025). Automated code assessment and feedback: a comprehensive model for improved programming education. *IEEE Access*.
- [3] Yousef, M., Mohamed, K., Medhat, W., Mohamed, E. H., Khoriba, G., & Arafa, T. (2025). BeGrading: large language models for enhanced feedback in programming education. *Neural Computing and Applications*, 37(2), 1027-1040.
- [4] Xu, H., Liu, H., Wu, Y., Kang, X., Chen, X., & Liu, Y. (2025). Exploring the Potential and Limitations of Large Language Models for Novice Program Fault Localization. *Journal of Systems and Software*, 112731.
- [5] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020, November). Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020* (pp. 1536-1547).
- [6] Wang, J., Huang, M., Nie, Y., & Li, J. (2021, May). Static analysis of source code vulnerability using machine learning techniques: A survey. In *2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD)* (pp. 76-86). IEEE.