arXiv:2510.06187v1 [cs.SE] 7 Oct 2025

Automated Program Repair of Uncompilable Student Code

Griffin Pitts
wgpitts@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Aum Pandya*
apandya4@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Darsh Rank*
drank@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Tirth Bhatt
tjbhatt@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Muntasir Hoq
mhoq@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Bita Akram
bakram@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Abstract

A significant portion of student programming submissions in CS1 learning environments are uncompilable, limiting their use in student modeling and downstream knowledge tracing. Traditional modeling pipelines often exclude these cases, discarding observations of student learning. This study investigates automated program repair as a strategy to recover uncompilable code while preserving students' structural intent for use in student modeling. Within this framework, we assess large language models (LLMs) as repair agents, including GPT-5 (OpenAI), Claude 3.5 Haiku (Anthropic), and Gemini 2.5 Flash (Google), under high- and low-context prompting conditions. Repairs were evaluated for compilability, edit distance, and preservation of students' original structure and logic. We find that while all three LLMs are capable of producing compilable repairs, their behavior diverges in how well they preserve students' control flow and code structure, which affects their pedagogical utility. By recovering uncompilable submissions, this work enables richer and more comprehensive analyses of learners' coding processes and development over time.

1 Introduction

Intelligent tutoring systems and related student modeling approaches often rely on large volumes of student submissions to track learning and provide personalized support. Yet, a significant share of novice programs are frequently uncompilable due to syntax errors. These uncompilable submissions are often excluded because they lack an evaluable score or performance measure, with existing methods dependent on code that can be parsed or executed, such as for generating abstract syntax trees in structural analyses [3]. However, uncompilable code can contain important information about students' intermediate reasoning and knowledge states, informing more complete models of their learning. Excluding such data reduces coverage and discards potentially meaningful evidence of student understanding.

Automated Program Repair (APR) has been widely studied in software engineering and computing education as a means to fix erroneous code. In educational contexts, APR is used to provide students with repaired solutions or feedback on logical errors. Early systems such as DeepFix demonstrated the feasibility of large-scale repair using neural sequence models [2], while Koutcheme et al. [4] explored the use of large language models for code infilling, applying generative approaches to repair student programs and provide feedback on functional or semantic errors. Later work such as CEMR [9] further advanced semantic repair by learning contextual edit operations from student submissions. While APR approaches are effective for producing correct code, they can also overwrite the student's original approach. In the context of modeling students' knowledge for feedback and analysis, semantic rewrites risk distorting the very evidence of learning we seek to model.

Syntax-only repair offers a more targeted opportunity. Novice submissions often fail due to small surface-level mistakes such as missing semicolons, unmatched braces, or undeclared variables that can be resolved with minimal edits. Lightweight fixes can retain the student's original structure while making the code executable, thereby preserving the integrity of their learning trajectory. Prior work has explored such corrections through rule-based systems, for instance Takhar and Aggarwal [8], who inserted missing tokens to make student programs compilable, though their approach was limited to a narrow set of predefined error patterns. Extending this line of research, we explore how large language models perform syntax-only repair on uncompilable student code, examining how model type and prompting context influence repair outcomes.

2 Methodology

2.1 Dataset

We use a publicly available dataset sourced from the CodeWorkout platform [1], an online programming learning environment. CodeWorkout provides short, auto-graded Java exercises relating to topics such as logic, conditionals, loops, and arrays. The dataset contains 57,670 anonymized Java submissions from 368 students in a CS1 course offered at a U.S. university during the Spring 2019 semester. Of these submissions, 18,787 submissions were correct and 38,883 were incorrect. Among the incorrect submissions, 9,906 (approximately 25%) failed to compile.

2.2 Evaluation

To ensure the analysis captured a representative range of novice programming errors, two CodeWorkout problems were randomly selected from a total set of 50. The selected problems addressed logic, conditionals, loops, arrays, and string manipulation. From these problems, 100 uncompilable Java submissions were randomly sampled for analysis. Each program was then processed under six experimental conditions, combining three language models (GPT-5, Claude 3.5 Haiku, and Gemini 2.5 Flash) with two prompting

*All authors contributed equally to this research.

contexts (low and high), yielding a total of 600 repaired outputs for evaluation. In the low-context condition, models received only the uncompilable student code and instructions to perform syntax-only repair with minimal edits that preserved control flow, identifiers, and formatting. The high-context condition additionally provided the compiler message, problem statement, and few-shot examples of correct and incorrect repairs.

Evaluation Criteria. To assess repair quality, we evaluated each model's output across three measures: compilation success, edit distance, and human evaluation. Compilation success indicated whether the repaired code executed without syntax errors. Edit distance, computed using normalized Levenshtein distance, measured how closely the repair aligned with the original, uncompilable code.

For human evaluation, four experts independently annotated all repaired outputs for Structural Preservation (SP) and Logical Preservation (LP). SP was coded as 1 if the repaired code maintained the original control flow and 0 otherwise. LP was coded as 1 if the repair consisted only of syntactic edits, such as adding missing delimiters or correcting variable names, and 0 if the repair changed the logical or semantic structure of the code.

The annotation process was iterative: experts coded an initial subset (10% of the repaired programs) jointly to align on definitions, then proceeded independently, calculating Cohen's Kappa (κ) after each round. When agreement fell below 0.80, in line with [5], disagreements were resolved and the codebook was refined. Inter-rater agreement exceeding $\kappa = 0.80$ was achieved on the second round, after which the full set was coded and analysis conducted. Statistical analyses included chi-square tests of independence for categorical outcomes (compilation success, SP, and LP) and ANOVA for continuous measures (edit distance).

3 Results

Our aim was to examine how different large language models and prompting contexts influence the quality of syntax-only code repair. Compilation success rates were high and consistent across models, with GPT-5 achieving 98.5% (591/600), Claude 3.5 96% (576/600), and Gemini 2.5 95.5% (573/600) compilable repairs. There were no statistically significant differences in compilation rates among the models ($\chi^2(2, \text{N} = 600) = 3.21$, p = .201). Prompting condition also did not significantly affect compilability ($\chi^2(1, \text{N} = 600) = 0.21$, p = 0.649), indicating that all three models effectively produced compilable repairs even under low-context prompting.

Edit distance significantly differed by model ($F(2, 594) = 16.22$, $p < 0.001$), with GPT-5 producing the smallest average edits (11.4), followed by Gemini 2.5 (13.8) and Claude 3.5 (24.4). Prompting condition had no significant effect on edit distance ($F(1, 594) = 0.004$, $p = 0.95$), contrary to our hypothesis that providing additional context might encourage models to over-correct or attempt to solve the problem rather than perform syntax-only repairs.

Human evaluation showed significant differences among models in preserving students' original structure and logic. Structural Preservation (SP) differed by model ($\chi^2(2, \text{N} = 579) = 18.10$, $p < 0.001$), with GPT-5 maintaining control flow in 190 of 197 repairs (96.4%), compared to Gemini 2.5 (186 of 190; 97.9%) and Claude 3.5 (170 of 192; 88.5%). Logic Preservation (LP) also differed significantly ($\chi^2(2, \text{N} = 549) = 22.36$, $p < 0.001$), with GPT-5 showing the highest proportion of logic-preserving repairs (166 of 192; 86.5%), followed by Gemini 2.5 (156 of 186; 83.9%) and Claude 3.5 (116 of 171; 67.8%). Prompting condition had no significant effect on either measure (SP: $\chi^2(1, \text{N} = 579) = 0.02$, $p = 0.88$; LP: $\chi^2(1, \text{N} = 549) = 0.00$, $p = 0.97$).

4 Discussion & Future Work

Our findings indicate that large language models can reliably perform repairs on short student code snippets, often conforming well to the explicit prompt instructions. However, similar to the tendencies observed by Řechtáčková et al. [7], the models occasionally strayed from the intended pedagogical scope by making stylistic or structural edits that went beyond minimal correction. This pattern reflects an ongoing challenge in achieving pedagogical alignment with LLMs used in educational settings [6].

While the present evaluation focused on compilation outcomes, edit distance, and expert annotation, more rigorous and fine-grained evaluation frameworks are needed to capture the pedagogical quality and instructional appropriateness of model-generated repairs. Future studies should incorporate larger and more diverse code samples and investigate how repaired submissions can be integrated into student modeling and intelligent tutoring system pipelines to better represent students' intermediate reasoning, misconceptions, and problem-solving progress. Further research should also consider pedagogical alignment more broadly, exploring how fine-tuning, context conditioning, and data augmentation can calibrate LLM behavior toward educational goals and ensure instructional consistency across learning contexts.

References

[1] Stephen H Edwards and Krishnan Panamalai Murali. 2017. CodeWorkout: short programming exercises with built-in data collection. In *Proceedings of the 2017 ACM conference on innovation and technology in computer science education.*

[2] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, Vol. 31.

[3] Muntasir Hoq, Griffin Pitts, Andrew Lan, Peter Brusilovsky, and Bita Akram. 2025. Pattern-based Knowledge Component Extraction from Student Code Using Representation Learning. *arXiv:2508.09281* (2025).

[4] Charles Koutcheme, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. 2023. Automated program repair using generative models for code infilling. In *International Conference on Artificial Intelligence in Education*. Springer, 798–803.

[5] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[6] Griffin Pitts, Anurata Prabha Hridi, and Arun-Balajiee Lekshmi-Narayanan. 2025. A Survey of LLM-Based Applications in Programming Education: Balancing Automation and Human Oversight. *arXiv preprint arXiv:2510.03719* (2025).

[7] Anna Řechtáčková, Alexandra Maximova, and Griffin Pitts. 2025. Finding misleading identifiers in novice code using LLMs. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 2.* 1595–1596.

[8] Rohit Takhar and Varun Aggarwal. 2019. Grading uncompilable programs. In *AAAI*, Vol. 33. 9389–9396.

[9] Han Wan, Hongzhen Luo, Mengying Li, and Xiaoyan Luo. 2024. Automated program repair for introductory programming assignments. *IEEE ToLT* (2024).