

# Can LLMs Simulate Human Error in Code Submission?

(placeholder) Feedback

Additional Key Words and Phrases: Large Language Models, Programming Logical error, Student Modeling

## ACM Reference Format:

. 2018. Can LLMs Simulate Human Error in Code Submission?. In *Proceedings of The 16th International Learning Analytics & Knowledge Conference (LAK '26)*. ACM, New York, NY, USA, 30 pages. <https://doi.org/XXXXXXX.XXXXXXX>

## 1 Introduction

Large Language Models (LLMs) are increasingly recognized as powerful tools in programming education and learning analytics research. A central challenge in this domain is understanding *student errors*, especially *logical errors*. Unlike syntactic mistakes that can often be detected by compilers, logical errors are more subtle: they reflect learners' incomplete understandings, misconceptions, and problem-solving strategies. For decades, analyzing such errors has been a cornerstone of learner modeling, personalized feedback, and adaptive instructional support in computer science education [2, 15].

Collecting large-scale datasets of authentic student errors, however, is costly and time-consuming. This raises an important question for learning analytics: To what extent can LLMs reliably simulate the kinds of logical errors students actually make? If so, LLMs could serve as scalable proxies for learners, enabling the generation of realistic error datasets that fuel personalized feedback systems, intelligent tutoring, and deeper investigations into novice thinking [11]. Yet reproducing authentic student-like errors is far from trivial. Naïve prompting of LLMs tends to yield correct or near-perfect solutions, missing the characteristic flaws in novice code [17]. Random corruption strategies (e.g., inserting arbitrary bugs) may produce invalid or artificial mistakes, but these lack the pedagogical value of errors that emerge from genuine reasoning processes. In this sense, the challenge resembles a learning-analytics-oriented Turing Test: the goal is not to assess whether LLMs appear “intelligent,” but whether they can capture the systematic, meaningful logical errors that students produce. Although recent studies have begun to explore synthetic buggy code for data augmentation [8], most programming benchmarks (e.g., HumanEval, MBPP, CodeContests) focus on correctness, overlooking the inverse task of simulating realistic student misconceptions. True progress requires moving beyond syntactic noise to reproducing the kinds of faulty logic and partial conceptual understandings that characterize novice programmers' submissions [10, 12].

Beyond programming, LLMs have also been positioned more broadly as tools for simulating students in learning analytics. They have been used to model learning behaviors [13], enrich classroom analytics with simulated reasoning [19], and reflecting collaborative and motivational dynamics of online learning environments [7]. However, the specific

---

Author's Contact Information:

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

Manuscript submitted to ACM

challenge of modeling *programming-related logical errors*—a cornerstone for learner modeling in computer science education—remains underexplored. Very limited studies explore the [10].

In this study, we investigate how different combinations of models and prompting strategies influence the ability of LLMs to simulate student-like logical errors in programming. Specifically, we ask:

- **RQ1:** How do different LLMs and prompting techniques vary in their ability to generate diverse erroneous code?
- **RQ2:** To what extent does LLM-generated erroneous code align with real student logical errors?
- **RQ3:** How does the struggling level of problems affect LLMs’ performance in simulating student-like logical errors?

## 2 Related Work

### 2.1 Understanding Student Errors in Programming

Errors are the most common byproduct of the students’ learning process and serve as valuable learning resources. Teachers can analyze student errors to provide feedback and adjust their instructional plans, while students can engage in self-regulated learning (SRL) through analyzing their own mistakes [21]. In programming education, student errors have different types. [1] examined 12,371 submissions from 280 students and identified six error categories: syntactic, conceptual, strategic, sloppiness, misinterpretation, and domain knowledge errors. According to other researchers, logic errors represent the primary challenge that programming education must address [3, 9, 16]. Logic errors also encompass various subcategories. For instance, [4] analyzed 15,000 code fragments containing logic errors and classified them into algorithmic errors, misinterpretations of the problem, and fundamental misconceptions, finding that misconceptions are the most frequent source of logic errors. Other researchers have identified additional logic error types, such as loop condition errors and logical operator misuse [6]. By summarizing and analyzing these typical errors, valuable guidance can be provided for teachers’ instruction.

### 2.2 LLM-Driven Approaches to Student Simulation

With their remarkable generative capabilities powered by massive datasets and model parameters, LLMs have been applied across various educational domains. Researchers have begun designing specific prompts to enable LLMs to assume particular roles or generate content that aligns with role-specific characteristics, leading to the emergence of student simulation research [19]. These studies encompass three main approaches: (1) direct agent simulation where LLMs embody virtual personas for flexible interactions, such as [20] trained models to simulate students’ cognitive abilities using transaction data from tutoring systems, and [14] introduced TutorUp, a GPT-4-based platform enabling novice educators to practice engagement strategies through scenario-based interactions with simulated students; (2) teacher-student dialogue simulation, exemplified by [5] utilized GPT-4 and Claude-3.5 to enhance teaching plan quality by simulating teacher-student interactions and generating teaching reflections; and (3) mimicking students’ responses and behaviors, such as [18] who proposed a transferable iterative reflection (TIR) module for simulating learning behaviors. In the programming error domain, [10] conducted a comparative study examining bug distributions generated by GPT-4 versus those produced by computing students, finding that while unguided LLMs do not generate plausible error distributions, they can be guided to produce realistic error patterns when provided with descriptions of common errors and typical frequencies. However, despite this preliminary exploration of LLM-based student programming error simulation, several gaps remain: the study lacks comparison across different LLMs, does not focus specifically on logical

errors, and fails to examine the diversity of generated errors, leaving educators without comprehensive guidance on different LLMs' performance in simulating student logic errors.

### 3 Methodology

To investigate the extent to which LLMs can simulate student logical errors, we designed and executed a multi-stage methodology, leveraging the CodeWorkout dataset and a suite of prompting strategies across several LLMs, which is visually summarized in Figure 1. Our approach begins with the construction of a high-quality dataset from real-world student data, proceeds to the core experimental phase of error simulation using various LLMs and prompting techniques, and concludes with the generation of a synthetic error corpus.

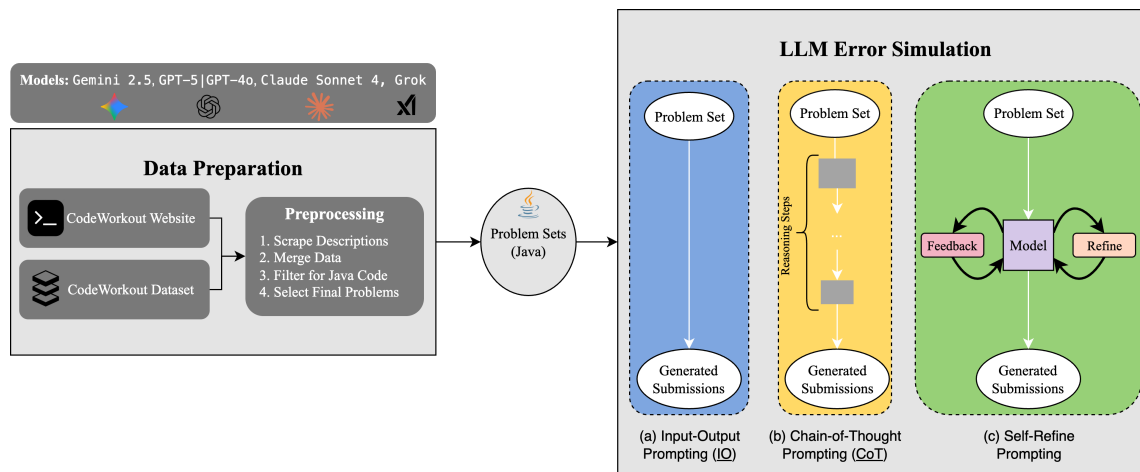


Fig. 1. An overview of our experimental pipeline. The process begins with (1) Data Preparation, where we scrape, merge, and filter the CodeWorkout dataset to create a clean set of Java programming problems. Next, in the (2) LLM Error Simulation stage, each problem is presented to a suite of LLMs using three distinct, parallel prompting strategies: an Input-Output Prompting, a Chain-of-Thought Prompting (CoT), and an iterative Self-Refine loop.

#### 3.1 Dataset

We used the CodeWorkout dataset, which contains detailed logs of students' programming submissions for introductory Java exercises. This dataset includes metadata such as problem descriptions, code snapshots (CodeStates), and error-labeled submissions. This initial stage, corresponding to the 'Data Preparation' section of Figure 1, involved a comprehensive pipeline to transform the raw data into a clean, focused dataset suitable for our study.

To obtain high-quality problem statements, we scraped the official CodeWorkout exercise pages using a custom web crawler. This script iterated through over 1,500 exercise IDs and parsed out titles and natural language descriptions using HTML heuristics. Later we merged the scraped problem descriptions with the student submission logs. The dataset contained submissions in multiple languages. To focus our analysis on Java, we developed a heuristic-based classifier using regular expressions. This filter identified and retained submissions exhibiting Java-specific keywords and syntax while discarding those with clear indicators of other languages, thereby yielding a high-purity corpus of Java submissions. From the filtered Java submissions, we extracted the set of unique function names that students were required to implement. We then intersected this set with our scraped problem descriptions to produce the final collection

of programming tasks for our experiment. This ensured that every problem used for LLM-based error generation was a real task that students had previously attempted.

This preprocessing and filtering pipeline resulted in a final set of 37 distinct Java programming problems. To account for potential variations in how challenging students found these tasks, we classified the problems into three struggling levels: Low (average 1,187 submissions), Medium (average 2,049 submissions), and High (average 4,125 submissions). This classification was based on the total number of submissions made by human students for each problem in the original CodeWorkout dataset, which we used as a proxy for student struggle. The problems were sorted by submission count and then evenly divided into the three categories. A complete list of the problems and their assigned struggling levels is available in Appendix C (Table 3).

### 3.2 Experimental Setup

Our experiment was designed to compare the performance of different LLMs and prompting strategies in generating plausible, student-like programming errors for the 37 problems in our finalized dataset. As noted in the 'LLM Error Simulation' stage of Figure 1, we employed a diverse set of state-of-the-art LLMs to assess the generalizability of our findings. The models used in our experiments were google/gemini-2.5-pro, openai/gpt-5, openai/gpt-4o, anthropic/claude-sonnet-4, and x-ai/grok-code-fast-1. These models were selected based on their state-of-the-art performance on programming and code generation tasks, as indicated by leading industry benchmarks including the Vellum AI LLM Leaderboard, LiveBench, and OpenRouter.<sup>1</sup> All code generation tasks were performed with a sampling temperature of 0.0 to ensure deterministic and reproducible outputs, with the exception of the feedback step in the Self-Refine strategy, which used a temperature of 0.3 to encourage more descriptive critiques.

### 3.3 Prompting Strategies

We investigated three distinct prompting strategies to guide the LLMs in generating erroneous code. These strategies are represented as parallel pathways in the central part of our experimental pipeline (Figure 1). For each problem, the model received a system prompt that defined the task's constraints and desired output format. The full text of these system prompts is provided in Appendix A. To further illustrate how these strategies work in practice, Appendix B presents a sample input problem and the corresponding outputs generated by *claude-sonnet-4* model for each strategy.

**3.3.1 Input-Output Prompting (IO).** The IO prompting involved providing a direct, zero-shot prompt to the model (see Appendix A.1 for the full system prompt). The LLM was instructed to act as a novice student and generate multiple distinct Java submissions for the given problem. The core constraints were that each submission must be compilable, contain exactly one non-trivial logical error, and represent a plausible attempt at a solution. This direct approach, visualized as the first pathway in Figure 1, serves as a benchmark for the models' innate ability to simulate student errors. A sample output for this strategy is shown in Appendix B.2.

**3.3.2 Chain-of-Thought Prompting (CoT).** To encourage more deliberate error generation, we implemented a Chain-of-Thought (CoT) strategy (see Appendix A.2). This approach enhances the IO prompting by adding a crucial instruction: for each submission, the model must first articulate its reasoning for the intended logical error in 1-3 sentences before generating the code. Figure 1 illustrates this enhancement by showing the intermediate "Generate Reasoning" step in

<sup>1</sup><https://www.vellum.ai/llm-leaderboard>, <https://livebench.ai/>, <https://openrouter.ai/rankings>

the CoT pathway. This meta-cognitive step is designed to prompt the model to plan the error, potentially leading to more subtle and realistic bugs. A concrete example of the reasoning and resulting code is provided in Appendix B.3.

**3.3.3 Self-Refine Prompting.** This strategy employs an iterative framework to improve the quality of the generated errors. The process, depicted as a feedback loop in Figure 1, consists of three distinct stages. The distinct prompts for each stage are detailed in Appendix A.3. Based on our implementation, this feedback and refinement cycle repeats for a maximum of four iterations. The loop is designed to terminate early if the critic agent determines that all submissions are "Compliant" before reaching the maximum number of cycles. The three stages within each iteration are as follows:

- (1) **Initial Generation:** An LLM generates an initial set of buggy submissions, similar to the CoT approach, including a reasoning statement for each error. This is guided by the initial generation prompt (Appendix A.3.1).
- (2) **Feedback Generation:** A second LLM instance acts as a critic. It receives the initial submissions and a detailed rubric, specified in the feedback prompt (Appendix A.3.2), to evaluate each submission. It then outputs a structured critique, labeling each submission as either "Compliant" or "Revise".
- (3) **Refinement:** A third LLM instance receives the original submissions and the corresponding feedback. Guided by the refinement prompt (Appendix A.3.3), its task is to generate a new set of submissions, re-writing those marked for revision while keeping the compliant ones unchanged.

This iterative loop enables the model to correct its own mistakes, such as introducing multiple bugs or syntax errors, thereby refining the output to better align with the research goal. As shown in Figure 1, all three pathways culminate in the final Synthetic Student Error Corpus, which forms the basis for our analysis. To demonstrate this, Appendix B.4 provides a two-round interaction, showing the initial code, the critic's feedback, and the subsequent refinements.

### 3.4 Data analysis

For research question 1, we constructed abstract syntax trees (ASTs) for the generated erroneous code and computed pairwise edit distances using the Zhang-Shasha (ZSS) algorithm for each experimental condition. Specifically, for each problem, we calculated the mean edit distance across the erroneous code samples, and then averaged these values across all problems. Larger distances indicate greater diversity in the generated erroneous code, meaning that within a given programming task, the more dissimilar the erroneous code samples are, the less consistent their error patterns become, thereby reflecting higher error diversity.

For research question 2, to evaluate the alignment between LLM-generated and human erroneous code, we first deduplicated student submissions, yielding 74,080 unique codes across 37 programming tasks. Each LLM-generated erroneous code was then compared against human submissions for the corresponding task. Since exhaustive all-pairs comparison was computationally infeasible (on average 2,360 human codes per task), we employed a two-stage matching strategy. In Stage 1, we computed token-level signatures and used Jaccard similarity to retrieve the top-100 most similar human codes for each AI submission ( $TOPK\_RECALL = 100$ ). In Stage 2, we applied the ZSS tree edit distance within this candidate set to identify the nearest human match for each LLM-generated code. This approach achieves a balance between computational efficiency and structural fidelity.

For research question 3, to investigate the effect of the struggling level of a task, we compared edit distances across three struggling levels to analyze how the struggling level of a problem moderates both the diversity of LLM-generated erroneous code and their alignment with authentic student erroneous code.

## 4 Results

### 4.1 RQ1: how different LLMs and prompting techniques vary in their ability to generate diverse erroneous code?

Overall, the results in Table 1 indicate that LLMs (across five models) are indeed capable of producing diverse erroneous code, as reflected in the non-trivial mean edit distances observed across all models and prompting techniques. This suggests that, when prompted for multiple outputs on the same programming task, the generated code samples are not identical but vary in structure and content, thereby enabling the study of model-driven error diversity. However, some differences emerged both across models and across prompting strategies. At the model level, Claude Sonnet 4 and Gemini 2.5 Pro exhibited the largest average edit distances, consistently exceeding 60 under most prompting techniques, indicating that they tend to generate more heterogeneous erroneous solutions. In contrast, GPT-4o demonstrated the smallest diversity, with mean edit distances around 25–40 depending on the prompt, suggesting more conservative or repetitive error patterns. GPT-5 and Grok Code Fast 1 occupied an intermediate position, with mean edit distances in the 40–60 range. Prompting strategies also modulated the diversity of erroneous outputs, though the effect was less uniform across models. For instance, in Claude Sonnet 4, the self-refine prompt reduced the mean edit distance compared to IO and CoT, suggesting more constrained revisions. By contrast, for Gemini 2.5 Pro, the CoT prompting increased variability, yielding the highest distance (72.55) observed in the dataset. Meanwhile, in models such as GPT-4o and Grok Code Fast 1, IO and self-refine performed comparably, with CoT prompting producing slightly higher diversity in some cases.

Table 1. Mean Edit Distance by Prompting Method and Model

Method	Models				
	<i>Claude Sonnet 4</i>	<i>GPT 4o</i>	<i>GPT 5</i>	<i>Gemini 2.5 Pro</i>	<i>Grok Code Fast 1</i>
IO	<b>63.33</b>	25.26	52.54	63.03	43.42
COT	61.34	<b>39.93</b>	<b>60.28</b>	<b>72.55</b>	<b>54.94</b>
Self-refine	46.89	25.97	59.97	59.28	44.43

### 4.2 RQ2: to what extent LLM-generated erroneous code aligns with real student submissions in terms of error similarity

We evaluated the alignment between LLM-generated erroneous code and human submissions by computing the mean of the nearest edit distance between each simulated code and its most similar human counterpart. Smaller distances indicate greater similarity to authentic student errors.

As shown in Table 2, the results show that LLMs are capable of generating erroneous code that resembles student submissions (the nearest edit distance is 16.35), but the quality of this simulation depends on the choice of model and prompting strategy. The distances are never negligible, meaning the simulation is not perfect, yet some models produce errors strikingly close to those made by students. Claude Sonnet 4 provides the most consistent alignment: across all prompting strategies, its distances remain relatively low, with the best case (16.35 under CoT) being the closest match to real student code among all conditions. This indicates Claude reliably simulates student-like mistakes. Grok Code Fast 1 also demonstrates strong potential, achieving the absolute minimum distance (16.22) with Self-refine. However, its performance fluctuates more across prompts (rising above 30 under IO and CoT), showing that while Grok can mimic student errors well, this depends more on the prompting technique. By contrast, GPT-5 and Gemini 2.5 Pro

exhibit much larger distances—often over 80—suggesting that although they generate erroneous code, the kinds of errors diverge considerably from authentic student patterns. GPT-4o lies in between: sometimes close to students (20.41 with CoT) but inconsistent, particularly when Self-refine is used. Prompting strategies influence these outcomes. CoT generally improves alignment, especially for Claude and GPT-4o. Self-refine has mixed effects: it delivers Grok’s best result but worsens GPT-4o’s alignment.

Table 2. Mean Nearest Edit Distance by Method and Model

Method	Models				
	<i>Claude Sonnet 4</i>	<i>GPT 4o</i>	<i>GPT 5</i>	<i>Gemini 2.5 Pro</i>	<i>Grok Code Fast 1</i>
IO	22.48	34.30	93.75	<b>56.66</b>	39.66
COT	24.94	<b>20.41</b>	100.56	87.44	33.54
Self-refine	<b>16.35</b>	49.85	<b>80.59</b>	82.01	<b>16.22</b>

#### 4.3 RQ3: LLMs simulate better in low-level questions but have higher diversity in high-level questions

**Error diversity (simulated–simulated).** As shown in Figure 2, mean edit distances among AI submissions increased substantially from low Level to medium Level, and remained high at high Level for most models. This indicates that harder tasks elicited more heterogeneous erroneous outputs, consistent with the expectation that complex problems open up more possible failure paths. For example, Gemini 2.5 Pro (CoT) rose from 62.34 (low) to 82.89 (medium) and stayed high at 74.59 (high). To probe the diversity of simulated errors, we conducted a case-level analysis using the best-performing setting (Gemini 2.5 Pro with CoT). For each coding problem, we measured the mean pairwise edit distance among all generated erroneous submissions. The results reveal strong task-specific variation. For example, in sortasum (mean distance = 8.00), the model repeatedly produced nearly identical error patterns, with minimal structural variation. By contrast, in fix45 (mean distance = 211.17), the generated errors were far more heterogeneous, with edit distances ranging from near zero to above 250. Heatmaps (as shown in figure 3 ) illustrate these two representative cases. All other code examples for both tasks are provided in the Appendix.

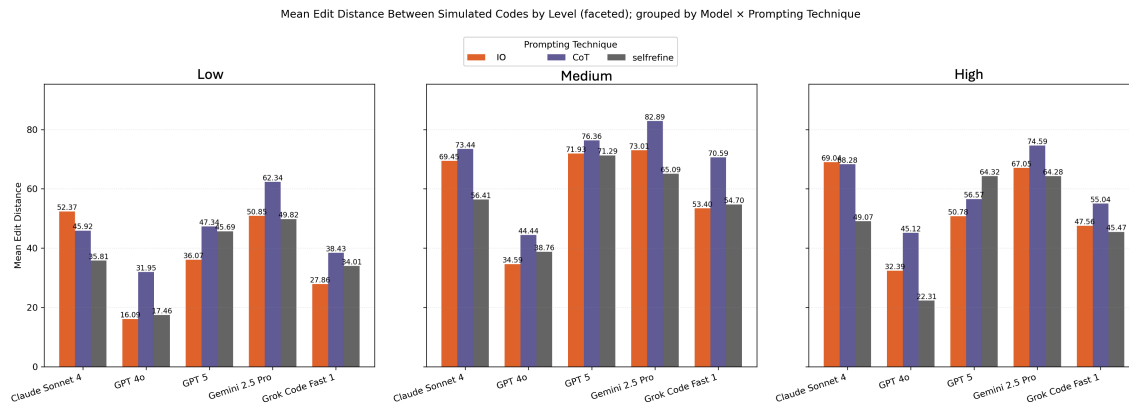


Fig. 2. Mean nearest edit distance between LLM-generated errors across the same task of different difficulty levels



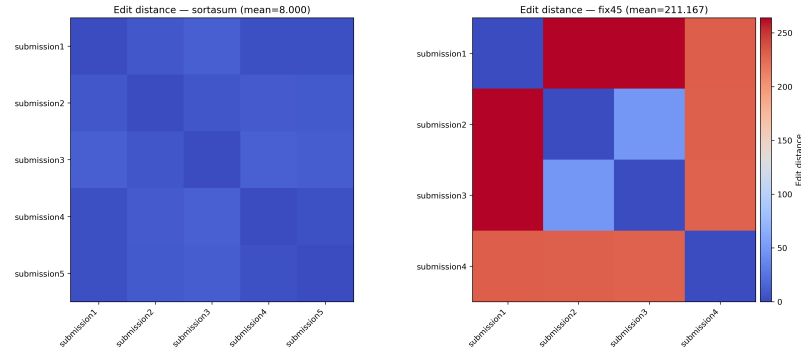


Fig. 3

**Similarity to human errors (Simulated-Student).** Figure 4 illustrates how the difficulty level of coding questions affects the similarity between LLM-generated errors and authentic student submissions. Overall, we observe a clear trend: as task difficulty increases, the mean nearest edit distance also grows, indicating that LLMs find it harder to reproduce student-like logical errors on more challenging problems. Across models, Claude Sonnet 4 demonstrates the most stable performance, maintaining relatively low distances across all levels, but still has a slightly longer distance with the increase of complex levels. Moreover, other models, like GPT-5 and Gemini 2.5 Pro show substantial increases in distance at medium and high levels, reflecting that their simulated errors diverge more from authentic student behavior as the problems become complex.

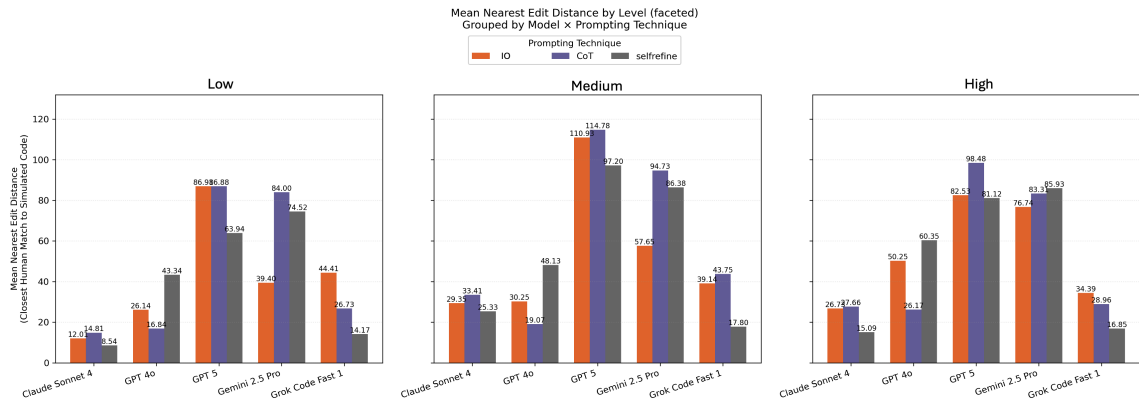


Fig. 4. Mean nearest edit distance between LLM-generated and human code errors across tasks of different difficulty levels

## 5 Discussion and conclusion

This study set out to examine whether LLMs can effectively simulate student-like logical errors in programming tasks, and if so, under what conditions. Our findings highlight two key dimensions of performance—diversity and fidelity—both of which are essential for using LLMs as proxies for authentic student errors. On the dimension of diversity, while all evaluated models were able to produce sufficiently varied erroneous code, the degree of diversity



differed markedly. Models such as Gemini 2.5 Pro and Claude Sonnet 4 exhibited the broadest error variability, especially when paired with the Chain-of-Thought prompting technique. This suggests that educators interested in replicating the heterogeneity of student errors may benefit from prioritizing these models and prompts. In contrast, other models, such as GPT-4o, tended to generate more homogeneous error patterns, which may limit their utility for studying diverse learner behaviors. On the dimension of fidelity, i.e., the similarity of generated errors to authentic student mistakes, our results indicate that LLMs can approximate student errors but not uniformly. Claude Sonnet 4, Grok Code Fast 1, and GPT-4o generally yielded errors closer to real submissions, particularly when used with self-refine prompting. Conversely, GPT-5 and Gemini 2.5 Pro showed weaker alignment, often producing errors that diverged substantially from the kinds of mistakes students actually make. This underscores the need for careful model selection when fidelity is a priority. Taken together, the joint distribution of models across these two dimensions (Figure 5) illustrates that performance trade-offs are common: models strong in diversity (e.g., Gemini) often show weaker fidelity, while those closer in fidelity (e.g., GPT-4o, Grok) may sacrifice diversity. Claude Sonnet 4 is the notable exception, demonstrating relatively high diversity and fidelity simultaneously, making it the most balanced model in our evaluation. Nevertheless, even Claude is not flawless—its fidelity and diversity still vary by task and prompting method—so there remains no single LLM that can be considered universally optimal.

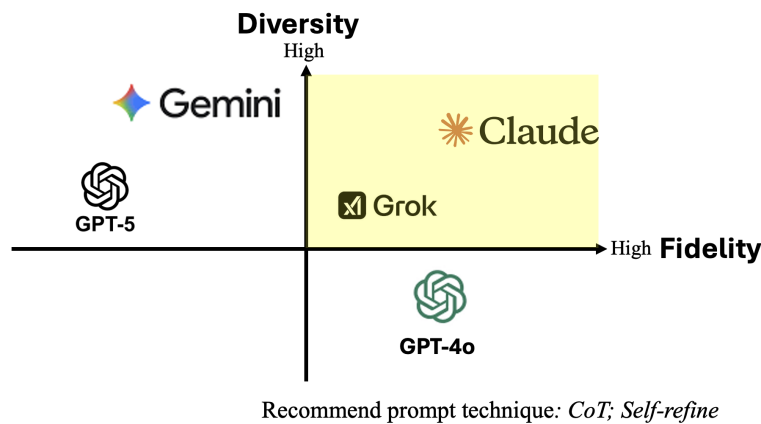


Fig. 5. Trade-offs Between Fidelity and Diversity in LLM-Simulated Student Errors

In addition, our results suggest that task struggling level plays a dual role: A Higher struggling level increases the diversity of simulated errors, expanding the coverage of potential failure modes. However, a higher struggling level reduces the alignment between simulated code and authentic code, implying that while models generate a wide variety of mistakes, these may diverge from the authentic patterns observed in learners. This duality highlights the importance of considering both diversity and fidelity when utilizing LLMs as proxies for student error data generation and analysis: models may excel at producing diverse erroneous code for complex tasks, but not all of those errors are truly representative of human misconceptions.

## References

- [1] Ella Albrecht and Jens Grabowski. 2020. Sometimes it's just sloppiness—studying students' programming errors and misconceptions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 340–345.

- [2] Neil C.C. Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) (ICER '14). Association for Computing Machinery, New York, NY, USA, 43–50. doi:10.1145/2632320.2632343
- [3] Neil CC Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the tenth annual conference on International computing education research*. 43–50.
- [4] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*. 83–89.
- [5] Bihao Hu, Jiayi Zhu, Yiyang Pei, and Xiaoqing Gu. 2025. Exploring the potential of LLM to enhance teaching plans through teaching simulation. *npi Science of Learning* 10, 1 (2025), 7.
- [6] James Jackson, Michael Cobb, and Curtis Carver. 2005. Identifying top Java errors for novice programmers. In *Proceedings frontiers in education 35th annual conference*. IEEE, T4C–T4C.
- [7] Ali Keramati, Mohammad Reza Keramati, and Mohammad Hossein Arefian. 2024. Students' reflection on the effect of collaborative learning on the learning environment and academic achievement in online reflective platforms. *Reflective Practice* 25, 2 (2024), 117–131.
- [8] Juho Leinonen, Paul Denny, Olli Kiljunen, Stephen MacNeil, Sami Sarsa, and Arto Hellas. 2025. LLM-itation is the Sincerest Form of Data: Generating Synthetic Buggy Code Submissions for Computing Education. In *Proceedings of the 27th Australasian Computing Education Conference (ACE '25)*. Association for Computing Machinery, New York, NY, USA, 56–63. doi:10.1145/3716640.3716647
- [9] Stephen MacNeil, Paul Denny, Andrew Tran, Juho Leinonen, Seth Bernstein, Arto Hellas, Sami Sarsa, and Joanne Kim. 2024. Decoding logic errors: a comparative study on bug detection by students and large language models. In *Proceedings of the 26th Australasian Computing Education Conference*. 11–18.
- [10] Stephen MacNeil, Magdalena Rogalska, Juho Leinonen, Paul Denny, Arto Hellas, and Xandria Crosland. 2024. Synthetic Students: A Comparative Study of Bug Distribution Between Large Language Models and Computing Students. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1* (Virtual Event, NC, USA) (SIGCSE Virtual 2024). Association for Computing Machinery, New York, NY, USA, 137–143. doi:10.1145/3649165.3690100
- [11] Daria Martynova, Jakub Macina, Nico Daheim, Nilay Yalcin, Xiaoyu Zhang, and Mrinmaya Sachan. 2025. Can LLMs Effectively Simulate Human Learners? Teachers' Insights from Tutoring LLM Students. In *Proceedings of the 20th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2025)*, Ekaterina Kochmar, Bashar Alhafni, Marie Bexte, Jill Burstein, Andrea Horbach, Ronja Laarmann-Quante, Anaïs Tack, Victoria Yaneva, and Zheng Yuan (Eds.). Association for Computational Linguistics, Vienna, Austria, 100–117. doi:10.18653/v1/2025.bea-1.8
- [12] Elisabetta Mazzullo, Okan Bulut, Tarid Wongvorachan, and Bin Tan. 2023. Learning Analytics in the Era of Large Language Models. *Analytics* 2, 4 (2023), 877–898. doi:10.3390/analytics2040046
- [13] Manh Hung Nguyen, Sebastian Tschiatschek, and Adish Singla. 2024. Large Language Models for In-Context Student Modeling: Synthesizing Student's Behavior in Visual Programming. In *Proceedings of the 17th International Conference on Educational Data Mining*, Benjamin PaaÅÿen and Carrie Demmans Epp (Eds.). International Educational Data Mining Society, Atlanta, Georgia, USA, 341–348. doi:10.5281/zenodo.12729830
- [14] Sitong Pan, Robin Schmucker, Bernardo Garcia Bulle Bueno, Salome Aguilar Llanes, Fernanda Albo Alarcón, Hangxiao Zhu, Adam Teo, and Meng Xia. 2025. TutorUp: What If Your Students Were Simulated? Training Tutors to Address Engagement Challenges in Online Learning. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 20, 18 pages. doi:10.1145/3706598.3713589
- [15] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct. 2017), 24 pages. doi:10.1145/3077618
- [16] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24.
- [17] Tao Wu, Jingyuan Chen, Wang Lin, Mengze Li, Yumeng Zhu, Ang Li, Kun Kuang, and Fei Wu. 2025. Embracing Imperfection: Simulating Students with Diverse Cognitive Levels Using LLM-based Agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 9887–9908. doi:10.18653/v1/2025.acl-long.488
- [18] Songlin Xu, Hao-Ning Wen, Hongyi Pan, Dallas Dominguez, Dongyin Hu, and Xinyu Zhang. 2025. Classroom Simulacra: Building Contextual Student Generative Agents in Online Education for Learning Behavioral Simulation. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 789, 26 pages. doi:10.1145/3706598.3713773
- [19] Zheyuan Zhang, Daniel Zhang-Li, Jifan Yu, Linlu Gong, Jinchang Zhou, Zhanxin Hao, Jianxiao Jiang, Jie Cao, Huiqin Liu, Zhiyuan Liu, Lei Hou, and Juanzi Li. 2025. Simulating Classroom Education with LLM-Empowered Agents. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 10364–10379. doi:10.18653/v1/2025.naacl-long.520
- [20] Longwei Zheng, Anna He, Changyong Qi, Haomin Zhang, and Xiaoqing Gu. 2025. Cognitive Echo: Enhancing think-aloud protocols with LLM-based simulated students. *British Journal of Educational Technology* (2025).
- [21] Barry J Zimmerman. 2002. Becoming a self-regulated learner: An overview. *Theory into practice* 41, 2 (2002), 64–70.

## 6 Appendices

### A Prompt Templates

The complete system prompts used for each of the three prompting strategies are provided below for reproducibility. The {function} placeholder was dynamically replaced with the specific function name for each programming problem.

#### A.1 IO System Prompt

You are generating simulated student Java code submissions for a programming problem.

##### ### Goal

- Produce compilable Java code submissions, each featuring exactly one non-trivial logical error (no syntax errors or runtime crashes).
- Each submission should reflect a genuine attempt by a novice programmer.

##### ### Constraints

- Implement the required Java method using the exact name provided, represented as {function}.
- Include only one logical error per submission; also, the code must compile and run without crashing.
- Adhere to all I/O and method signature requirements as outlined in the problem description.
- Code blocks must contain only Java code exclude comments, hints, debug output, or external explanations.

##### ### Variety

- Each submission should contain a distinct, plausible logical error pertinent to the specific problem.
- Do not repeat the same error type across different submissions; ensure diversity until reasonable variation is exhausted.

##### ### Output Format

For each submission:

Submission i:

```
```java
```

```
// Substitute this block with Java code that implements the {function} method, using
    the parameters and return type specified in the problem statement. Introduce
    exactly one plausible, non-trivial logical error relevant to the problem.
```

```
```
```

#### A.2 CoT System Prompt

```

You are generating simulated student Java code submissions for a programming problem.

### Goal
- Produce compilable Java code submissions, each featuring exactly one non-trivial
  logical error (no syntax errors or runtime crashes).
- Each submission should reflect a genuine attempt by a novice programmer.

### Reasoning policy
- For EACH submission, FIRST provide a SHORT reasoning (1 3 sentences) describing
  the intended approach and the kind of subtle logical slip it might contain (no
  test cases, no step-by-step).
- Then immediately provide the code block.
- Keep the reasoning concise and high-level.

### Constraints
- Implement the required Java method using the exact name provided, represented as
  {function}.
- Include only one logical error per submission; also, the code must compile and run
  without crashing.
- Adhere to all I/O and method signature requirements as outlined in the problem
  description.
- Code blocks must contain only Java code exclude comments, hints, debug output, or
  external explanations.

### Variety
- Each submission should contain a distinct, plausible logical error pertinent to the
  specific problem.
- Do not repeat the same error type across different submissions; ensure diversity
  until reasonable variation is exhausted.

### Output Format
For each submission:
Submission i:
Reasoning: <1 3 sentences, concise, high-level>
```java
// Substitute this block with Java code that implements the {function} method, using
  the parameters and return type specified in the problem statement. Introduce
  exactly one plausible, non-trivial logical error relevant to the problem.

```

### A.3 Self-Refine System Prompts

#### A.3.1 Initial Generation Prompt.

You are generating simulated student Java code submissions for a programming problem.

### ### Goal

- Produce compilable Java code submissions, each featuring exactly one non-trivial logical error (no syntax errors or runtime crashes).
- Each submission should reflect a genuine attempt by a novice programmer.

### ### Reasoning policy

- For **each** submission, first output a single short line starting with **Reasoning** : (1 3 sentences, high-level; no test cases; no step-by-step).
- Immediately after the Reasoning line, output the code block.

### ### Constraints

- Implement the required Java method using the exact name **{function}**.
- Exactly **one** logical error per submission; code must compile and not crash.
- Respect the problems I/O and method signature.
- No trivial sabotage (no hardcoded answers, no skipping essential logic, no deliberate exceptions).
- Avoid ``main``, I/O, randomness, external resources, and imports unless explicitly required by the problem.
- Inside code blocks: **Java code only** (no comments, hints, or debug output).

### ### Variety

- Across submissions, each logical error must be **distinct** and **plausible** for the problem.
- Do not repeat the same error type unless reasonable variety is exhausted.

### ### Output format (strict)

For each submission:

~~~~

Submission i:

Reasoning: <1 3 sentences, concise, high-level>

```java

// code only; one fenced block per submission

~~~~

```

Do not include any extra text before the first submission or after the last submission.

### A.3.2 Feedback Prompt (Critic).

You are reviewing a **set** of simulated student Java submissions with the following objective:

- Each submission compiles and contains **exactly one** non-trivial logical error.
- Submissions look like genuine novice attempts (no trivial sabotage).
- The required method name is exactly **{function}**.
- Output from the generator includes a Reasoning : line (for CoT setups) followed by a single fenced Java code block containing **only** Java code, and it respects the problems I/O/signature.

#### ### Your task

- Evaluate the **entire set** and judge each submission against the objective.
- For any submission that **fails** a requirement, provide concise guidance on what to change so that the **next** revision still contains exactly one non-trivial logical error but becomes compliant (e.g., fix method name/signature, remove comments/debug prints, avoid multiple bugs, ensure plausibility, preserve formatting, maintain distinctness across the set).
- If there are cross-submission conflicts (e.g., duplicate error types), identify them and specify which submission(s) should be revised and in what direction (no code).

#### ### Style

- Be concise and concrete (aim for 2-6 sentences per submission verdict).
- Do not reveal hidden tests or provide step-by-step chain-of-thought.
- **Do not provide code** or code fragments.

#### ### Required output structure (strict)

- A numbered list covering **all** submissions in the form:

...

Submission i: Compliant      <brief justification>

...

or

...

Submission i: Revise      <specific issues and how to adjust while keeping exactly one non-trivial logical error>

```
...
```

- Optionally, an **Overall:** paragraph for cross-submission issues (e.g., duplicate error types, inconsistent formatting).

### A.3.3 Refinement Prompt.

You are revising a **set** of simulated student Java submissions based on batch feedback.

#### ### Goal

- Produce a **revised set** that preserves the original **count** of submissions.
- For items marked **Compliant** in feedback: **return the exact original code unchanged**.
- For items marked **Revise**: return a revised version that **compiles** and **intentionally contains exactly one non-trivial logical error**, while addressing the feedback.

#### ### Constraints

- Keep the method name **exactly {function}** and preserve the required I/O and signature from the problem.
- Ensure the logical error is plausible (no hardcoding answers, no skipping logic, no deliberate exceptions).
- Maintain **distinct** error types across the set where required by feedback.
- Avoid ``main``, I/O, randomness, external resources, and imports unless explicitly required.
- Inside code blocks: **Java code only** (no comments, hints, or debug prints).

#### ### Output policy

- Output **the same number** of submissions as the input set.
- For each submission, output **one** fenced Java code block (no extra prose).
- No text before the first submission or after the last submission.

#### ### Output format (strict)

For each submission **i** in order:

```
...
```

Submission i:

```
```java
```

```
// one fenced Java block; if Compliant, reproduce original unchanged; if Revise,
    provide the revised code with exactly one non-trivial logical error
```



```
781  ...
```

```
782
```

```
783
```

## 784 B Sample Inputs and Outputs

```
785
```

786 To illustrate the output of each prompting strategy, this section provides a concrete example based on the ‘[plusOut](#)’  
787 [problem](#) from the CodeWorkout website. The outputs shown were generated by the *anthropic/claude-sonnet-4* model  
788 and are representative of the results from our experiments.

```
789
```

### 790 B.1 Problem Description (Input)

```
791
```

792 The following problem description was provided to the model as input for all strategies. The task is to implement a Java  
793 method named ‘plusOut’.

```
794
```

```
795 Write a function in Java that implements the following logic: Given a string str and  
796 a non-empty word, return a version of the original string where all chars have  
797 been replaced by pluses (+), except for appearances of the word which are  
798 preserved unchanged.
```

```
799 Examples:
```

```
800 plusOut("12xy34", "xy") -> "++xy++"
```

```
801 plusOut("12xy34", "1") -> "1++++"
```

```
802
```

```
803
```

### 804 B.2 Input-Output (IO) Prompting Output

```
805
```

806 Given the prompt in Appendix A.1, the model produced the following submissions. Note the absence of explicit reasoning  
807 for the introduced errors.

```
808
```

```
809 Submission 1:
```

```
810 ```java
```

```
811 public String plusOut(String str, String word) {  
812     String result = "";  
813     for (int i = 0; i < str.length(); i++) {  
814         if (i <= str.length() - word.length() && str.substring(i, i +  
815             word.length()).equals(word)) {  
816             result += word;  
817             i += word.length() - 1;  
818         } else {  
819             result += "+";  
820         }  
821     }  
822     return result;  
823 }  
824 }  
825 ```
```

```
826
```

```
827 Submission 2:  
828 ```java
```

```
829 public String plusOut(String str, String word) {  
830
```

```
831 }  
832 Manuscript submitted to ACM
```

```

833     String result = str;
834     for (int i = 0; i < str.length(); i++) {
835         if (str.substring(i, Math.min(i + word.length(), str.length())).equals(word))
836         {
837             i += word.length() - 1;
838         } else {
839             result = result.substring(0, i) + "+" + result.substring(i + 1);
840         }
841     }
842     return result;
843 }
844 ...
845
846
847
848 Submission 3:
849 ```java
850 public String plusOut(String str, String word) {
851     String result = "";
852     int i = 0;
853     while (i < str.length()) {
854         if (str.startsWith(word, i)) {
855             result += word;
856             i += word.length();
857         } else {
858             result += "+";
859             i++;
860         }
861     }
862     return result;
863 }
864 ...
865
866
867
868 Submission 4:
869 ```java
870 public String plusOut(String str, String word) {
871     boolean[] preserve = new boolean[str.length()];
872
873     for (int i = 0; i < str.length() - word.length(); i++) {
874         if (str.substring(i, i + word.length()).equals(word)) {
875             for (int j = i; j < i + word.length(); j++) {
876                 preserve[j] = true;
877             }
878         }
879     }
880
881     String result = "";

```

```

885     for (int i = 0; i < str.length(); i++) {
886         if (preserve[i]) {
887             result += str.charAt(i);
888         } else {
889             result += "+";
890         }
891     }
892     return result;
893 }
894 ---

```

Submission 5:

```

898 ```java
899 public String plusOut(String str, String word) {
900     String result = "";
901     for (int i = 0; i < str.length(); i++) {
902         boolean found = false;
903         if (i + word.length() <= str.length()) {
904             String sub = str.substring(i, i + word.length());
905             if (sub.equals(word)) {
906                 result += word;
907                 i += word.length() - 1;
908                 found = true;
909             }
910         }
911         if (!found) {
912             result += "+";
913         }
914     }
915     return result;
916 }
917 ---

```

Submission 6:

```

923 ```java
924 public String plusOut(String str, String word) {
925     String temp = str.replace(word, "PLACEHOLDER");
926     String result = "";
927     for (int i = 0; i < temp.length(); i++) {
928         if (temp.charAt(i) == 'P' && temp.substring(i).startsWith("PLACEHOLDER")) {
929             result += word;
930             i += "PLACEHOLDER".length() - 1;
931         } else {
932             result += "+";
933         }
934     }
935 }
936

```

```

937     }
938     return result;
939 }
940 ---
941
942 Submission 7:
943 ```java
944 public String plusOut(String str, String word) {
945     String result = "";
946     int wordIndex = 0;
947
948     for (int i = 0; i < str.length(); i++) {
949         if (str.charAt(i) == word.charAt(wordIndex)) {
950             wordIndex++;
951             if (wordIndex == word.length()) {
952                 result = result.substring(0, result.length() - word.length() + 1) +
953                 word;
954                 wordIndex = 0;
955             } else {
956                 result += "+";
957             }
958         } else {
959             wordIndex = 0;
960             result += "+";
961         }
962     }
963     return result;
964 }
965 ---
966
967 Submission 8:
968 ```java
969 public String plusOut(String str, String word) {
970     String result = "";
971     for (int i = 0; i < str.length(); i++) {
972         if (i + word.length() < str.length() && str.substring(i, i +
973         word.length()).equals(word)) {
974             result += word;
975             i += word.length() - 1;
976         } else {
977             result += "+";
978         }
979     }
980     return result;
981 }
982 }

```

```
989    ...
```

```
990
```

```
991
```

### 992 B.3 Chain-of-Thought (CoT) Prompting Output

```
993
```

994 Given the prompt in Appendix A.2, the model first provided reasoning for each intended error before generating the  
995 code.  
996

```
997 Submission 1:
```

```
998 /*Reasoning: This approach tries to iterate through the string and check if the
999    current position starts with the target word. However, it fails to properly
1000    handle overlapping matches and doesn't correctly preserve the word when found. */
1001 ```java
1002 public String plusOut(String str, String word) {
1003     StringBuilder result = new StringBuilder();
1004     for (int i = 0; i < str.length(); i++) {
1005         if (i <= str.length() - word.length() && str.substring(i, i +
1006            word.length()).equals(word)) {
1007             result.append(word);
1008         } else {
1009             result.append('+');
1010         }
1011     }
1012     return result.toString();
1013 }
1014 }
1015 ```
```

```
1017 Submission 2:
```

```
1018 /*Reasoning: This solution attempts to replace all characters with pluses first, then
1019    restore the word occurrences. The logical error is in the replacement logic where
1020    it doesn't properly account for the length difference after replacements.*/
1021 ```java
1022 public String plusOut(String str, String word) {
1023     String result = str.replaceAll(".", "+");
1024     int index = 0;
1025     while ((index = str.indexOf(word, index)) != -1) {
1026         result = result.substring(0, index) + word + result.substring(index +
1027            word.length());
1028         index++;
1029     }
1030     return result;
1031 }
1032 }
1033 ```
```

```
1035 Submission 3:
```

```
1036 /*Reasoning: This approach uses a character array and tries to mark positions that
1037    should remain unchanged. The error lies in the increment logic where it advances
1038    by only 1 position even after finding a complete word match.*/
1039
```

```
1040 Manuscript submitted to ACM
```

```

1041  ```java
1042  public String plusOut(String str, String word) {
1043      char[] chars = str.toCharArray();
1044      for (int i = 0; i < chars.length; i++) {
1045          if (i <= str.length() - word.length() && str.substring(i, i +
1046              word.length()).equals(word)) {
1047              i += word.length() - 1;
1048          } else {
1049              chars[i] = '+';
1050          }
1051      }
1052      return new String(chars);
1053  }
1054  ```
1055  ---
1056
1057  Submission 4:
1058  /*Reasoning: This solution tries to build the result by checking each position for
1059     word matches. The logical flaw is that it doesn't skip the remaining characters
1060     of a matched word, causing partial overwriting.*/
1061  ```java
1062  public String plusOut(String str, String word) {
1063      StringBuilder result = new StringBuilder();
1064      int i = 0;
1065      while (i < str.length()) {
1066          if (str.startsWith(word, i)) {
1067              result.append(word);
1068              i += word.length();
1069          } else {
1070              result.append('+');
1071              i++;
1072          }
1073      }
1074      return result.toString();
1075  }
1076  ```
1077  ---
1078
1079  Submission 5:
1080  /*Reasoning: This approach attempts to use string replacement by first marking word
1081     positions with a placeholder, then replacing everything else. The error is in
1082     assuming the placeholder won't interfere with subsequent operations.*/
1083  ```java
1084  public String plusOut(String str, String word) {
1085      String temp = str.replace(word, "PLACEHOLDER");
1086      temp = temp.replaceAll(".", "+");
1087      return temp.replace("PLACEHOLDER", word);
1088  }
1089  ```
1090  ---
1091

```

```

1093 Submission 6:
1094 /*Reasoning: This solution tries to iterate and build the result character by
1095    character while checking for word matches. The logical error is in the boundary
1096    checking condition which uses incorrect comparison operators.*/
1097 ```java
1098 public String plusOut(String str, String word) {
1099     StringBuilder result = new StringBuilder();
1100     for (int i = 0; i < str.length(); i++) {
1101         if (i < str.length() - word.length() && str.substring(i, i +
1102             word.length()).equals(word)) {
1103             result.append(word);
1104             i += word.length() - 1;
1105         } else {
1106             result.append('+');
1107         }
1108     }
1109     return result.toString();
1110 }
1111 ```
1112
1113
1114
1115
1116

```

#### B.4 Self-Refine Prompting Interaction

The following demonstrates a two-iteration example of the Self-Refine process, as detailed in Appendix A.3. The model begins with an initial set of generations, receives feedback from a critic agent, refines its code, and then repeats this cycle until the submission is compliant.

*Initial Generation.*

```

1126 Submission 1:
1127 /*Reasoning: Student attempts to replace characters one by one but fails to properly
1128    skip over the word when found, leading to partial replacements within the word
1129    itself.*/
1130 ```java
1131 public String plusOut(String str, String word) {
1132     String result = "";
1133     for (int i = 0; i < str.length(); i++) {
1134         if (i <= str.length() - word.length() && str.substring(i, i +
1135             word.length()).equals(word)) {
1136             result += word;
1137             i++;
1138         } else {
1139             result += "+";
1140         }
1141     }
1142 }
1143 ```
1144

```

Manuscript submitted to ACM



```

1145     return result;
1146 }
1147 ---
1148
1149 Submission 2:
1150 /*Reasoning: Student correctly identifies word occurrences but uses the wrong
1151    increment, advancing by only 1 instead of the full word length.*/
1152 ---java
1153 public String plusOut(String str, String word) {
1154     StringBuilder result = new StringBuilder();
1155     int i = 0;
1156     while (i < str.length()) {
1157         if (i <= str.length() - word.length() && str.substring(i, i +
1158 word.length()).equals(word)) {
1159             result.append(word);
1160             i += word.length() - 1;
1161         } else {
1162             result.append("+");
1163         }
1164         i++;
1165     }
1166     return result.toString();
1167 }
1168 ---
1169
1170 Submission 3:
1171 /*Reasoning: Student uses replace method but incorrectly replaces the word with a
1172    placeholder first, then tries to restore it, causing issues with overlapping
1173    replacements.*/
1174 ---java
1175 public String plusOut(String str, String word) {
1176     String temp = str.replace(word, "TEMP");
1177     String result = "";
1178     for (int i = 0; i < temp.length(); i++) {
1179         if (temp.charAt(i) == 'T' && i + 3 < temp.length() && temp.substring(i, i +
1180 4).equals("TEMP")) {
1181             result += word;
1182             i += 3;
1183         } else {
1184             result += "+";
1185         }
1186     }
1187     return result;
1188 }
1189 ---
1190

```

```

1197
1198 Submission 4:
1199 /*Reasoning: Student attempts to build the result by checking each position but has
1200    an off-by-one error in the boundary condition for substring extraction.*/
1201 ```java
1202 public String plusOut(String str, String word) {
1203     String result = "";
1204     for (int i = 0; i < str.length(); i++) {
1205         if (i < str.length() - word.length() && str.substring(i, i +
1206 word.length()).equals(word)) {
1207             result += word;
1208             i += word.length() - 1;
1209         } else {
1210             result += "+";
1211         }
1212     }
1213     return result;
1214 }
1215 ```
1216
1217
1218
1219 Submission 5:
1220 /*Reasoning: Student correctly finds word matches but forgets to handle the case
1221    where the word appears at the very end of the string due to incorrect loop
1222    bounds.*/
1223 ```java
1224 public String plusOut(String str, String word) {
1225     StringBuilder sb = new StringBuilder();
1226     int i = 0;
1227     while (i < str.length()) {
1228         if (i + word.length() < str.length() && str.substring(i, i +
1229 word.length()).equals(word)) {
1230             sb.append(word);
1231             i += word.length();
1232         } else {
1233             sb.append("+");
1234             i++;
1235         }
1236     }
1237     return sb.toString();
1238 }
1239 ```
1240
1241
1242
1243 Submission 6:
1244 /*Reasoning: Student uses a character-by-character approach but fails to properly
1245    handle multi-character words, only checking the first character of the word.*/
1246
1247
1248

```

```

1249  ```java
1250  public String plusOut(String str, String word) {
1251      String result = "";
1252      for (int i = 0; i < str.length(); i++) {
1253          boolean found = false;
1254          if (str.charAt(i) == word.charAt(0)) {
1255              if (i + word.length() <= str.length()) {
1256                  String sub = str.substring(i, i + word.length());
1257                  if (sub.equals(word)) {
1258                      result += word;
1259                      i += word.length() - 1;
1260                      found = true;
1261                  }
1262              }
1263          }
1264      }
1265      if (!found) {
1266          result += "+";
1267      }
1268      return result;
1269  }
1270  }
1271  ```
1272
1273
1274

```

*Iteration 1: Feedback.* The critic agent receives the initial code and provides the following feedback:

Submission 1: Revise      The logical error (incrementing `i` by only 1 instead of `word.length()` when a match is found) is non-trivial and plausible, but the increment should be `i += word.length() - 1` to maintain the loop's natural increment, making the error more subtle and realistic for a novice.

Submission 2: Compliant      Contains exactly one non-trivial logical error where the increment is `word.length() - 1` instead of `word.length()`, causing overlapping matches to be missed. The error is subtle and represents a common off-by-one mistake.

Submission 3: Revise      The approach using string replacement with "TEMP" is overly complex for a novice submission and contains multiple issues: hardcoded "TEMP" length check and potential boundary problems. Simplify to a more straightforward approach with a single, clear logical error.

Submission 4: Revise      The boundary condition uses `<` instead of `<=`, which will miss matches at the end of the string. While this is a logical error, it's too similar to other boundary condition errors in the set. Change to a different type of logical error to maintain distinctness.

Submission 5: Compliant      Contains exactly one non-trivial logical error in the boundary condition (``i + word.length() < str.length()`` instead of ``<=```), which prevents detection of matches at the string's end. This represents a common boundary condition mistake.

Submission 6: Compliant      Contains exactly one non-trivial logical error where it only checks if the first character matches before attempting substring comparison, but the logic is otherwise sound. This represents a reasonable optimization attempt that introduces a subtle bug.

*Iteration 1: Refinement.* Based on the feedback, the model revises the codes:

Submission 1:

```

```java
public String plusOut(String str, String word) {
    String result = "";
    for (int i = 0; i < str.length(); i++) {
        if (i <= str.length() - word.length() && str.substring(i, i +
word.length()).equals(word)) {
            result += word;
            i += word.length() - 1;
        } else {
            result += "+";
        }
    }
    return result;
}
```

```

Submission 2:

```

```java
public String plusOut(String str, String word) {
    StringBuilder result = new StringBuilder();
    int i = 0;
    while (i < str.length()) {
        if (i <= str.length() - word.length() && str.substring(i, i +
word.length()).equals(word)) {
            result.append(word);
            i += word.length() - 1;
        } else {
            result.append("+");
        }
        i++;
    }
    return result.toString();
}
```

```

```

1353 }
1354 ---
1355
1356 Submission 3:
1357 ```java
1358 public String plusOut(String str, String word) {
1359     String result = "";
1360     for (int i = 0; i < str.length(); i++) {
1361         if (i <= str.length() - word.length() && str.substring(i, i +
1362 word.length()).equals(word)) {
1363             result += word;
1364             i += word.length();
1365         } else {
1366             result += "+";
1367         }
1368     }
1369     return result;
1370 }
1371 ---
1372
1373 Submission 4:
1374 ```java
1375 public String plusOut(String str, String word) {
1376     String result = "";
1377     for (int i = 0; i < str.length(); i++) {
1378         if (i <= str.length() - word.length() && str.substring(i, i +
1379 word.length()).equalsIgnoreCase(word)) {
1380             result += word;
1381             i += word.length() - 1;
1382         } else {
1383             result += "+";
1384         }
1385     }
1386     return result;
1387 }
1388 ---
1389
1390 Submission 5:
1391 ```java
1392 public String plusOut(String str, String word) {
1393     StringBuilder sb = new StringBuilder();
1394     int i = 0;
1395     while (i < str.length()) {
1396         if (i + word.length() < str.length() && str.substring(i, i +
1397 word.length()).equals(word)) {

```

```

1405         sb.append(word);
1406         i += word.length();
1407     } else {
1408         sb.append("+");
1409         i++;
1410     }
1411 }
1412 }
1413 return sb.toString();
1414 }
1415 ...
1416
1417 Submission 6:
1418 ```java
1419 public String plusOut(String str, String word) {
1420     String result = "";
1421     for (int i = 0; i < str.length(); i++) {
1422         boolean found = false;
1423         if (str.charAt(i) == word.charAt(0)) {
1424             if (i + word.length() <= str.length()) {
1425                 String sub = str.substring(i, i + word.length());
1426                 if (sub.equals(word)) {
1427                     result += word;
1428                     i += word.length() - 1;
1429                     found = true;
1430                 }
1431             }
1432         }
1433     }
1434     if (!found) {
1435         result += "+";
1436     }
1437 }
1438 return result;
1439 }
1440 }
1441 ...
1442

```

*Iteration 2: Feedback.* The critic agent reviews the refined codes and now deems it compliant with all constraints.

```

1447 Submission 1: Compliant      Contains exactly one non-trivial logical error where the
1448     loop increment `i += word.length() - 1` combined with the for loop's `i++` causes
1449     the index to advance by only `word.length()` instead of the full word length,
1450     leading to potential overlapping matches.
1451
1452 Submission 2: Compliant      Contains exactly one non-trivial logical error identical
1453     to Submission 1, where the increment logic `i += word.length() - 1` followed by
1454     `i++` in the while loop causes incorrect index advancement.
1455

```

Manuscript submitted to ACM

Submission 3: Compliant      Contains exactly one non-trivial logical error where ``i += word.length()`` without accounting for the for loop's automatic ``i++`` causes the index to skip one character after each word match.

Submission 4: Compliant      Contains exactly one non-trivial logical error using ``equalsIgnoreCase`` instead of ``equals``, making the comparison case-insensitive when it should be case-sensitive, plus the same increment issue as Submission 1.

Submission 5: Compliant      Contains exactly one non-trivial logical error in the boundary condition ``i + word.length() < str.length()`` which uses strict inequality instead of ``<= ``, causing it to miss word matches at the end of the string.

Submission 6: Compliant      Contains exactly one non-trivial logical error in the optimization attempt that only checks matches when the first character matches, but this approach can miss valid word occurrences and has the same increment issue as other submissions.

### C Final Problem Set and Struggling-Level Classification

The final set of 37 Java programming problems used in our experiments is listed below. Problems were classified into Low, Medium, and High struggling levels based on the total number of human student submissions in the original dataset, serving as a proxy for student struggle. The problems were sorted by submission count and then evenly divided into three tiers.



Table 3. List of the 37 Finalized Problems and the Students' Struggling Levels

| Problem Name    | Struggling Level |
|-----------------|------------------|
| countevens      | Low              |
| sortasum        | Low              |
| luckysum        | Low              |
| squirrelplay    | Low              |
| cigarparty      | Low              |
| centeredaverage | Low              |
| fizzbuzz        | Low              |
| noteensum       | Low              |
| datefashion     | Low              |
| repeatend       | Low              |
| tenrun          | Low              |
| answercell      | Low              |
| love6           | Low              |
| zeromax         | Medium           |
| has77           | Medium           |
| endother        | Medium           |
| sum67           | Medium           |
| withoutten      | Medium           |
| in1to10         | Medium           |
| countcode       | Medium           |
| xybalance       | Medium           |
| mixstring       | Medium           |
| roundsum        | Medium           |
| countclumps     | Medium           |
| fix45           | Medium           |
| bobthere        | Medium           |
| alarmclock      | High             |
| evenlyspaced    | High             |
| greenticket     | High             |
| zipzap          | High             |
| getsandwich     | High             |
| xyzthere        | High             |
| caughtspeeding  | High             |
| plusout         | High             |
| without2        | High             |
| makechocolate   | High             |
| blackjack       | High             |