Decoding Logic Errors: A Comparative Study on Bug Detection by Students and Large Language Models

Stephen MacNeil
Temple University
Philadelphia, PA, United States
stephen.macneil@temple.edu

Paul Denny
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Andrew Tran
Temple University
Philadelphia, PA, United States
andrew.tran10@temple.edu

Juho Leinonen
University of Auckland
Auckland, New Zealand
juho.leinonen@auckland.ac.nz

Seth Bernstein
Temple University
Philadelphia, PA, United States
seth.bernstein@temple.edu

Arto Hellas
Aalto University
Espoo, Finland
arto.hellas@aalto.fi

Sami Sarsa
Aalto University
Espoo, Finland
sami.sarsa@aalto.fi

Joanne Kim
Temple University
Philadelphia, PA, United States
joanne.kim@temple.edu

ABSTRACT

Identifying and resolving logic errors can be one of the most frustrating challenges for novices programmers. Unlike syntax errors, for which a compiler or interpreter can issue a message, logic errors can be subtle. In certain conditions, buggy code may even exhibit correct behavior – in other cases, the issue might be about how a problem statement has been interpreted. Such errors can be hard to spot when reading the code, and they can also at times be missed by automated tests. There is great educational potential in automatically detecting logic errors, especially when paired with suitable feedback for novices. Large language models (LLMs) have recently demonstrated surprising performance for a range of computing tasks, including generating and explaining code. These capabilities are closely linked to code syntax, which aligns with the next token prediction behavior of LLMs. On the other hand, logic errors relate to the runtime performance of code and thus may not be as well suited to analysis by LLMs. To explore this, we investigate the performance of two popular LLMs, GPT-3 and GPT-4, for detecting and providing a novice-friendly explanation of logic errors. We compare LLM performance with a large cohort of introductory computing students ($n = 964$) solving the same error detection task. Through a mixed-methods analysis of student and model responses, we observe significant improvement in logic error identification between the previous and current generation of LLMs, and find that both LLM generations significantly outperform students. We outline how such models could be integrated into computing education tools, and discuss their potential for supporting students when learning programming.

ACE 2024, January 29–February 02, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-1619-5/24/01.
https://doi.org/10.1145/3636243.3636245

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

large language models, generative AI, programming errors, bug detection, computing education

ACM Reference Format:
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 *Australian Computing Education Conference (ACE 2024), January 29–February 02, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3636243.3636245

1 INTRODUCTION

Learning to program involves navigating a landscape where mistakes are an inherent part of the journey. Novice programmers are bound to encounter numerous errors when writing code, ranging from logic flaws and syntactical inaccuracies to runtime glitches. These mistakes pose substantial hurdles to students as they strive to develop their programming skills. Despite extensive efforts by computing education researchers and practitioners to establish taxonomies and recognize patterns of common programming errors [1, 6, 37, 38, 61], the process of effectively detecting and resolving bugs remains a persistent challenge.

Simultaneously, the emergence of large language models (LLMs) has demonstrated remarkable capabilities in understanding and generating text that is highly similar to the text generated by people. These models, trained on vast amounts of textual data, have been used in a variety of computing education contexts including helping students to understand code [30, 35] and programming error messages [31]. These use cases demonstrate the ability of LLMs to understand the syntax and structure of code. Still, it is unclear whether models can reason about runtime performance without explicitly running the code. Therefore, detecting runtime

errors may present a challenge for LLMs, limiting their potential to help learners.

In this paper, we conduct a large-scale comparative study that investigates the abilities of two LLMs and students to detect bugs in faulty code. We recruited 964 students in a large introductory C programming class to identify bugs in three code examples. The selected code examples contained three types of bugs including an out-of-bounds error, an expression error, and an operator error. Students were selected because they are increasingly relying on LLMs as a legitimate help-seeking resource [20, 25, 62]. Our results suggest that LLMs outperform students in bug detection performance, especially for faulty code. However, in addition to detecting the pre-inserted bugs, the LLMs had a tendency to be overly proactive, also commenting on extremely minor 'bugs' such as naming conventions, and other considerations that might be overwhelming if used for learning purposes. GPT-4 was nearly perfect at identifying bugs in faulty code, but was much more likely than GPT-3 to identify these minor 'bugs' in the correct programs and therefore performed 'poorly' on correct code. Studying correct code was important because students may use these tools when their code is mostly correct, and a list of minor errors may be demoralizing or may lead them off-track. Based on our findings, we conclude that LLMs appear to be capable of identifying logic errors, outperforming students at this task. However, additional work is needed to extend this work toward more complex code examples and with more advanced computing students. Given that experts are more likely to 'chunk' code and see emergent structure, it is unclear whether they would be more or less able to identify bugs in the code without writing test cases.

2 RELATED WORK

2.1 Students and Bugs in Code

Bugs and errors are a common feature in student code and understanding the encountered problems and errors has been a long-standing endeavor within Computing Education Research. Early research in this area centered often on specific problems such as the looping problem or the rainfall problem [24, 48, 51, 52], leading also toward investigations into the design and features of programming languages (e.g. [51]). In general there are differences in frequency of programming errors [53] and the time that it takes to fix those errors [7, 11, 38, 50]. The types of errors that students encounter also gradually change [2], and they can stem from multiple sources [2, 14]. These sources include misinterpreting the programming problem and having flaws in programming knowledge [14], not to mention the role of the used programming language [26].

When students encounter a problem, they need to resolve it. Resolving programming problems – or debugging – can be done using multiple approaches, including tracing code, commenting out code, and adding print statements [17, 40, 59]. Simply looking at the code and trying to find places that do not look right – i.e. pattern matching – can also be a viable strategy in some cases [17]. Like programming, finding problems in code by tracing the code is a skill, and both of them have been highlighted as something that students can struggle with. As an example, an ITiCSE working group from 2001 highlighted a lack of programming skills at the end of introductory programming course [39], and a subsequent ITiCSE working group from 2004 focused on the results by looking into students' ability to read and trace code [32], also highlighting problems. These issues have in part led to national and international efforts in understanding the struggles that students face, such as the BRACElet project that started in 2004 [58].

These studies tend to highlight that students have difficulties with tracing code [32, 56], which might in part be explainable by lack of expertise. A student might, when solving a tracing problem, even just guess a solution if they do not have a higher-level reasoning strategy [32], or might simply have misconceptions about how a program executes, which in turn leads to faulty conclusions [56]. This possibility of guessing code tracing outcomes has also in part led to the emergence of "explain in plain English" problems. For these problems, students are expected to provide a high-level overview of the program's functionality and purpose rather than simply outlining how the program executes [33, 60]. These problems can also be challenging, and any tools that would help students learn to understand and explain code would be of benefit.

2.2 Generative AI and Computing Education

Recently computing education researchers are expressing concern and excitement about the ways that generative models may affect the computing education landscape [12, 29, 34, 36, 41, 42, 62]. While a strong consensus about how we should adapt our pedagogical practice has yet to emerge, each of these discussions acknowledge that generative models are not likely a passing fad.

Numerous examples of the capabilities of generative models are emerging such as their ability to both solve and create programming assignments [16, 44], explain code [30, 35], identify programming concepts [55], answer multiple choice questions [46, 47], write code [43, 57], solve visual problems [21], and enhance programming error messages [31]. These use cases are critical because without understanding the capabilities of generative models, it is extremely challenging to adapt to this rapidly changing landscape.

However, limited work has investigated the capabilities of generative models to identify bugs within code. Given that novice programmers often encounter bugs and may lack the ability to identify and fix these bugs, it is important to explore the capabilities of generative models to accomplish this task. Very recent papers focus on enhancing programming error messages [31] and automatically repairing bugs in code [15, 23, 28]. In this paper, we add to the growing set of use cases by exploring the potential for generative models to identify potential bugs and errors.

3 METHOD

3.1 Research Questions

Previous research has demonstrated many impressive capabilities of large language models. However, many of these examples, such as generating explanations and identifying programming concepts, are closely linked to code syntax, which aligns with the next token prediction behavior of LLMs. To better explore the potential limits of LLMs, this study focuses on identifying logic errors in code, which relate to the runtime performance of code, and thus may not be as well suited to analysis by LLMs as they are unable to execute code. If large language models perform well in this task, there is an exciting opportunity to use these models to help students to debug

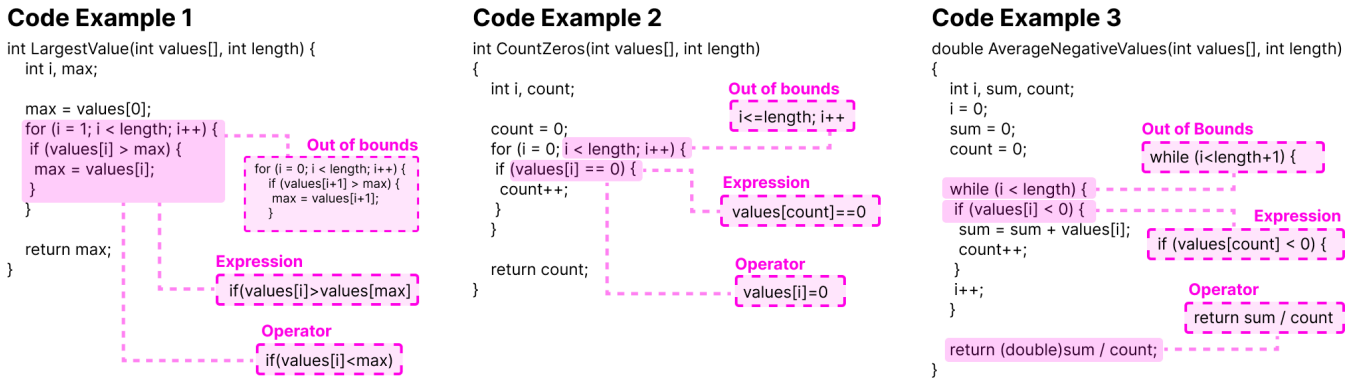

Figure 1: The three examples of code with the correct variant, and the three incorrect variants annotated. The incorrect variants were 1) operator error, 2) expression error, and 3) out-of-bounds error.

their code. Based on these goals, we investigated the following research questions:

RQ 1: How do students and large language models compare in their ability to correctly identify logic errors in faulty code?

RQ 2: Which types of logic errors are easiest for students and large language models to correctly identify?

RQ 3: How many bugs or issues do students and large language models identify when reviewing faulty and correct code?

3.2 Study Design

In this study, we seek to investigate the performance of large language models in detecting bugs in faulty code. We conducted a study that compared the performance of students with the two large language models GPT-3 and GPT-4. Performance was measured across three code examples with four variants. These variants included the correct code and three variants with bugs introduced: 1) an operator error, 2) an out-of-bounds error, and 3) an expression error. The study was designed with two between-subjects components which include the source of the detection method, i.e., whether it was performed by the students, GPT-3, or GPT-4, and the bug variant. The study also included a within-subjects component which was the three code examples. By showing students multiple examples, we could partially control for participant error.

3.2.1 Participants, Data Collection, and Ethics. The data used in this study were collected from a first-year C programming course at the University of Auckland. The data were collected during a single lab session that ran over a one-week period. Leading up to this lab, the course covered the concepts of arithmetic, types, functions, loops, and arrays. We collected 964 total complete responses from students. The data collection followed the ethical guidelines of the university and was approved by the University of Auckland Human Participants Ethics Committee[1].

3.2.2 Study Tasks. As part of the lab, students were shown the three code examples in Figure 1. Each example contains a function with a single loop that processes elements of an array. The task for the students was to identify any bugs that might exist within the code. The instructions said *"Consider the following definition of a function called <Function Name>:"* which was followed by the code without comments. They were then asked to come up with a short description of what they believe the intended purpose of the function to be. This was followed by having them *"List all errors, if any, found in this code based on your explanation of the purpose of the function. It is possible that the code contains one or more small errors (however, this is not necessarily true and the code may be correct). If you can identify any errors in the implementation of the code, you should describe these errors."*

3.2.3 Measures. The data collection resulted in 2980 total responses from students. In addition, 30 LLM responses were generated for each code example and version pair by varying the temperature and prompt to account for variations that might affect performance. This resulted in 720 total additional responses from the two models.

A team of four researchers manually coded each student and model response. The coders evaluated the **correctness** of the identified bug as a dichotomous variable (e.g.: correct or incorrect). The coders also evaluated the **number of bugs** that the response contained. The coding was mutually exclusive: a response correctly identifying a bug but also noting other incorrect bugs was coded as correct. When coding the example that did not contain bugs, we coded a blank response or an explicit statement that no bugs were contained as a correct response and other responses were considered incorrect. This coding scheme did not allow for explicitly tracking false positives and false negatives, but it was necessary to obtain substantial inter-rater reliability ($\kappa = 0.873$, 30 ratings). Students often did not explicitly state the bug so we coded their response as 'correct' even if they only provided a solution that would fix the expected bug.

3.2.4 Analysis for Conditional Differences. We analyzed the dependent measures (e.g.: number of bugs) using a linear mixed-effects model. The main fixed factors of interest were the "Source" (representing GPT-3, GPT-4, or Students) and the "Version" of the code example (representing different versions of the example). Additionally, an interaction term between "Source" and "Version" was included to examine potential differences in bug identification across sources and versions. To account for potential dependencies among

[1]Reference number UAHPEC25279.

observations from the same example, a random intercept term was included in the model specification. This random effect was nested within the "Code Example" factor, capturing the variability associated with different examples. Pairwise comparisons were made using the Tukey method with Holm's correction for multiple comparisons.

3.3 Models

3.3.1 Model Specification. To automatically identify the bugs in the study, we used two large language models [8] developed by OpenAI. The first model, *text-davinci-003*, has been widely used up until the time of running the study. Later, when GPT-4 was released, we included results using the *gpt-4-0314* model to understand how the state-of-the-art models perform at the same task.

3.3.2 Prompt Engineering. Prompt engineering is a process of developing instructions to guide the responses of an LLM. The specificity and phrasing of these prompts have the potential to strongly influence the content and quality of the responses [3, 49, 63]. Understanding the potential effects that prompts can have on performance, we used multiple prompting strategies to account for this aspect. In addition, the hyperparameters of an LLM, such as the temperature, can also affect the output. Lower temperatures tend to result in more deterministic responses while higher temperatures tend to provide more 'creative' responses. We chose to use the default temperature of 0.7 and a lower temperature of 0.3. The three prompts used for this study are listed below.

- # List all errors and bugs, if any, found in the following C code: <code>
- # List any issues, including bugs, errors, or potential problems that exist in the following C code: <code>
- # Assume the role of a highly intelligent computer scientist who is capable of easily finding bugs and errors by reading source code. List all errors and bugs, if any, found in the following C code: <code>

Between the variations in prompt and temperature, there were 6 possible permutation. For each permutation, we issued 5 requests to the OpenAI API. The reason for issuing 5 requests was to account for the non-deterministic nature of LLM prompts. This resulted in 30 responses for each combination of code example and bug type and 360 total requests to OpenAI.

4 RESULTS

4.1 Bug Detection Performance

Performance in bug detection rates varied between the students and the models, as shown in Table 1. GPT-3 exhibited an overall correctness rate of 85.3%, while GPT-4 closely followed with a correctness rate of 85.0%. Notably, students had a much lower bug detection rate at 49.1%. While both models detected bugs at nearly twice the rate of students, performance was even higher when only considering model performance on faulty code.

4.1.1 For faulty code, LLMs outperform students. When presented with incorrect code, GPT-3 exhibited a bug detection rate of 87.3%, demonstrating a substantial ability to identify coding errors. GPT-4 surpassed this performance with an impressive bug detection rate of 99.2%, indicating a higher sensitivity to identifying bugs within faulty code. On the other hand, students detected bugs at rate of 34.5%, showcasing a limited proficiency in detecting coding errors.

4.1.2 LLMs tended to identify bugs in correct code. In the case of identifying correctly functioning code, GPT-3 achieved a bug detection rate of 79.4% (i.e., classified the code as bug-free). GPT-4, however, displayed a comparatively lower rate of 42.2% in correctly identifying bug-free code. In contrast, students demonstrated a notably high proficiency in identifying correct code, with a bug detection rate of 92.8%.

4.2 Number of Bugs Detected

We observed statistically significant differences in the number of bugs identified by GPT-3, GPT-4, and students. The results of the linear mixed-effects model, which are summarized in Table 2, show that GPT-4 identified significantly more bugs than GPT-3 ($\beta = 0.76$, $SE = 0.13$, $z = 5.77$, $p < 0.001$) and students ($\beta = 1.70$, $SE = 0.11$, $z = -15.13$, $p < 0.001$). The model estimated that GPT-4 identified 0.761 more bugs than GPT-3 and 1.701 more bugs than students when other variables were held constant. GPT-3 also identified statistically significantly more bugs than students ($\beta = 0.94$, $SE = 0.11$, $z = -8.30$, $p < 0.001$).

4.3 Analyzing the Bug Reports

4.3.1 GPT-4 was more verbose, even when normalized by the number of bugs detected. We computed the average word count for responses made by students and each model. GPT-4 responses had on average 129.0 ($\sigma = 44.7$) words followed by GPT-3 and students with 54.2 ($\sigma = 19.5$) and 38.9 ($\sigma = 27.0$) words respectively. This constitutes a 3.31-fold increase in the number of words GPT-4 produced compared to students. Given the differences in number of bugs identified by source, we normalized word count by the number of bugs reported. This resulted in 52.7 ($\sigma = 25.7$) words for GPT-4 and 23.5 ($\sigma = 9.11$) and 35.6 ($\sigma = 24.5$) words for GPT-3 and students. These results should be contextualized by the observation that GPT-4 had a tendency to provide partial and in a few cases complete solutions for the bugs that it identified. Moreover, the models exhibited a more concentrated distribution around their means, while student responses exhibited notably higher variability.

4.3.2 Qualitative analysis of responses. To better understand the capabilities of large language models in detecting bugs in code and to draw distinctions between students' responses and these models, we did a qualitative exploration of the error messages. By analyzing selected bug reports generated by the models in our study, we could shed light on a variety of shortcomings of the models' approaches. As suggested by our analysis of word count, there appear to be differences in the ways that LLMs and students identify and describe bugs. Students tended to describe the bug, offer a solution, or both. We did not observe an instance where a model offered a solution without also describing the underlying bug. Models often combined a description of the bug along with a partial solution, in some cases they provided a complete solution. GPT-4 appeared to be most likely to offer a complete solution. As noted in the analysis of bug detection performance, LLMs often

Table 1: A summary of student and model performance in correctly identifying bugs. In instances where coders were unsure, they coded it as 'uncertain.' These are excluded from the table. For instance, if we consider the number of correct and incorrect responses for GPT-3 in Code Example 1 with Bug 2, their total does not sum to 30.

		Code Example 1				Code Example 2				Code Example 3			
Source		Bug 1	Bug 2	Bug 3	Correct	Bug 1	Bug 2	Bug 3	Correct	Bug 1	Bug 2	Bug 3	Correct
Student	correct	56	91	108	207	71	90	82	222	84	66	69	220
	incorrect	147	146	125	19	184	165	130	6	155	143	162	26
	rate	0.276	0.384	0.464	0.916	0.278	0.353	0.387	0.974	0.351	0.316	0.299	0.894
GPT-3	correct	30	23	8	30	29	27	24	29	29	30	24	12
	incorrect	0	4	22	0	0	0	1	1	0	0	6	17
	rate	1	0.852	0.267	1	1	1	0.96	0.967	1	1	0.800	0.414
GPT-4	correct	29	30	28	19	30	30	28	0	30	30	28	0
	incorrect	0	0	1	11	0	0	1	30	0	0	1	30
	rate	1	1	0.966	0.633	1	1	0.966	0	1	1	0.966	0

Table 2: Number of bugs detected by condition

Contrasts	Estimate	Std. Error	z value	Pr(>|z|)
GPT4 - GPT3	0.761	0.132	5.766	< 0.001 ***
GPT3 - Student	0.940	0.113	8.303	< 0.001 ***
GPT4 - Student	1.701	0.112	15.133	< 0.001 ***
Bug2 - Bug1	0.390	0.134	2.910	0.02724 *
Correct - Bug2	0.652	0.186	3.495	0.00389 **
Bug3 - Bug1	0.644	0.139	4.626	< 0.001 ***
Correct - Bug2	0.262	0.187	1.399	0.63514
Bug3 - Bug2	0.254	0.140	1.812	0.36131
Bug3 - Correct	-0.008	0.190	-0.040	1.00000

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
(Adjusted p values reported – Holm Correction)

identified bugs even for 'correct' code. Below is an example where GPT-4 identified 6 bugs in otherwise correct code for Example 2:

(1) No function prototype: There is no function prototype provided for the `CountZeros` function. It is a good practice to provide a function prototype before the function definition to avoid any potential issues with implicit declarations.
(2) No include statements: The code does not include any necessary header files, such as `stdio.h` or `stdlib.h`, which might be required for certain operations or functions used in the program.
(3) Inconsistent naming convention: The function name `COUNTZEROS` is mentioned in the question, but the actual function name in the code is `CountZeros`. It is important to maintain a consistent naming convention throughout the code.
(4) No error handling: The function does not perform any error handling, such as checking for NULL pointers or negative array length. This could lead to undefined behavior if the function is called with invalid arguments.
(5) No comments: The code lacks comments explaining the purpose of the function and its parameters. Adding comments can make the code more readable and easier to understand for others.
(6) Potential integer overflow: If the `length` parameter is very large.

It should be noted that most of these identified 'bugs' could be interpreted as areas for improvement. For example, the model suggests improvements to naming conventions or including comments. However, both the fourth and sixth bugs could actually result in runtime issues. The model identifies a lack of input sanitization checks for two potential corner cases. The qualitative investigation underscores the intricacies of LLMs' code interpretations. They exhibit detailed understanding but can occasionally miss the mark on context or offer feedback that, while technically correct, might feel misaligned with the coder's intent. Balancing LLM insights with human discernment could yield the most effective outcomes.

Many student responses just contained the proposed change without any explanation or reasoning. They often did not explicitly define a bug but instead only described the solution. Some students also indicated errors that either did not fix the issue, introduced new problems, or focused too heavily on syntactical correctness without addressing the core problem. In the example below, a student highlighted changes that should be made to the code which do not fix the bug:

- `i = 0;` should change to `i = 1;` to avoid using the 0th value.
- Instead of `count++`, use `count = count + 1;`.
- There should be no space between `for` and the opening parenthesis `(`.
- Similarly, there should be no space between `if` and the opening parenthesis `(`.

5 DISCUSSION

Our results suggest that large language models are more capable than students at identifying bugs in code. There are multiple possible explanations for this. First, more expert programmers often do not necessarily need to read the code character by character or word by word when forming an understanding of the code, rather, they study features of the code that are relevant to the task at hand [19]. Consequently, a student may miss syntax errors or minor bugs, if they are not in focus. This can also be explained by the happy path mentality where because most of the code is correct, students may

become complacent and fail to detect bugs; some bugs also take more time to identify and fix than others [11]. Participants were explicitly prompted to find errors, which puts them into an explicit debugging mindset. In practice, they might not critically examine their code with the same scrutiny, so the bug detection rate for students may actually be even lower in practice.

Both LLMs performed extremely well, with GPT-4 performing near perfect when presented with buggy code. However, both models performed poorly in our analysis of correct code as they identified very minor bugs and stylistic aspects such as naming conventions contrary to our expectation that they would classify the code as bug-free. While the suggestions were largely correct, it might not be helpful to point out minor bugs and code conventions in otherwise correct code, especially considering students' preferences for concise bug reports [13].

One noticeable difference between GPT-3 and GPT-4 was that GPT-4 would point out these minor bugs more than GPT-3. One possible explanation for this is that the newer model has possibly had more instruction fine-tuning, where the model is trained to follow instructions from the user. This might cause the model to try please the user by going above and beyond the ask, e.g. in our case not only pointing out the obvious bug, but also commenting on more minor issues. We also found that GPT-4 was more verbose, even when controlling for the number of bugs in the code. This aligns with prior findings where newer models often add superfluous textual content to responses [10] and may come up with non-existing bugs to fix when asked to help with buggy code [20].

The ability of LLMs to correctly identify bugs at a much higher rate than students has exciting implications for computing education. LLMs could be used to help novices (and more experienced programmers too) in detecting bugs in code, for example, by having LLMs integrated directly into the IDE that students use to work on their course exercises. Models could make suggestions for improvement as they did in cases with correct code or identify subtle logic errors in the code, potentially building on prior research on improving programming error messages, which has the promise of improving learning [5, 13]. Despite the allure of the technological possibilities, there likely should be a mechanism that would control how often the suggestions would be shown, as not all errors require help [20]. Similarly, it is important to carefully curate educational content, especially with growing concerns about over-reliance on LLMs [9, 29, 34, 57, 62]. To mitigate potential issues, it is likely preferable to avoid directly presenting errors and solutions to students. Instead, pedagogical systems could detect when students are spinning their wheels trying to debug their code [4] and then use the LLM to scaffold students toward identifying the error themselves. Thus providing learning opportunities that also mitigate stress associated with debugging.

Similarly, as LLMs are adept at detecting bugs and writing suggestions on how to fix them, they could be further integrated into teacher tools. As an example, tools such as OverCode [18] and CodeClusters [27] that are designed to provide feedback to masses of students could be integrated with LLMs so that LLMs would create draft feedback, which instructors then could – when needed – adjust and send out. The ability of LLMs to identify rare corner cases also has interesting implications for teaching testing, as feedback from LLMs could help with writing more comprehensive test suites. The good performance of the models could also lead to new, innovative exercise types. For example, we envision that an LLM could create buggy code where students would need to find and fix the bug – similarly, one activity could be trying to create bugs that LLMs fail to identify. Such activities could also provide additional data on learning, which then could be used to fine-tune LLMs.

As the educational landscape continues to adapt to LLMs [29, 41, 42, 62], the new bug capabilities of LLMs identified in this paper may further inform how students seek help in classroom settings [22].

5.1 Limitations

To make the task more ecologically valid, we provided students with an open-response question rather than a multiple-choice question. This had the advantage that students could not guess the right answer and was more similar to how students would encounter code in the wild; however, it became difficult to differentiate between a response that explicitly stated 'no bugs' and a blank response. To address this limitation, we evaluated the rates of default responses by variant and observed no statistically significant difference in the number of default responses across all four variants.

Participants were asked to identify any bugs that were present within the code, so in this case, a lack of an explicit response was treated as a default response (e.g.: 'no bugs'). To assess the impact on our results, we recalculated percentages by excluding blanks. The revised student correctness rates are as follows: 79.6% (133 blanks removed), 89.8% (169 blanks removed), and 60.0% (181 blanks removed). These results represent a conservative estimate, considering only explicitly stated correct answers. The resulting rates remained higher than GPT-4, but closer to GPT-3 correctness rates.

Participants were also explicitly instructed to identify bugs as part of the lab activity. While prior research has demonstrated that debugging others' code can be challenging [59], it is possible that if students were studying their own code, it might have been easier for them. Relatedly, the code did not have comments that would explain what each line of code does. This may align with code students often encounter naturally, but could have affected the students' performance or required the model to infer too much from the code structure and function name.

The code examples used in this study only contained a single intended error. It is possible that the presence of multiple bugs in code might affect the performance of LLMs (and students) in detecting bugs. The goal for this paper was an initial tightly scoped investigation of identifying a bug within code. Future work will investigate cases where multiple bugs are included.

In our study, we employed a robust approach by utilizing three distinct prompts, leveraging multiple models, including both GPT-3 and GPT-4, and exploring various temperatures (i.e., 0.4 and 0.7). Additionally, each prompt was issued multiple times to accommodate the inherent probabilistic nature of generative AI. While we acknowledge the potential impact of further prompt optimization on mitigating false positives in the correct code condition, it is essential to note the dynamic nature of these models, characterized by continuous changes in verbosity and performance [45, 46, 54]. Rather than providing a definitive characterization of performance, our primary objective was to delve into a novel capability of LLMs.

In this work, we report on the results of a study that compares the ability of students and large language models to identify bugs in faulty and correct code. Our results suggest that students struggled to find bugs in faulty code, but that they performed relatively well at identifying whether the code was correct. The models performed in the opposite way: both models (GPT-3 and GPT-4) strongly outperformed students in identifying bugs in faulty code, but tended to identify many minor 'bugs' which were more akin to suggestions for improvement when the code was correct. This suggests that models are overly sensitive toward discovering bugs in code. While some of the minor bugs detected by the models could be considered 'bugs', such over-sensitivity could be seen as a negative for integrating LLMs into teaching. If students receive superflous feedback on minor stylistic aspects, for example, they might start disregarding any useful feedback from the models too.

ACKNOWLEDGMENTS

We are grateful for the grant from the Ulla Tuominen Foundation to Juho Leinonen.

REFERENCES

[1] Basma S Alqadi and Jonathan I Maletic. 2017. An empirical study of debugging patterns among novices programmers. In *Proc. of the 2017 ACM SIGCSE technical Symp. on computer science education*. 15–20.

[2] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proc. of the 46th ACM Technical Symp. on Computer Science Education*. 522–527.

[3] Simran Arora, Avanika Narayan, Mayee F. Chen, Laurel J. Orr, Neel Guha, Kush S Bhatia, Ines Chami, Frederic Sala, and Christopher R'e. 2022. Ask Me Anything: A simple strategy for prompting language models. *ArXiv* abs/2210.02441 (2022).

[4] Joseph E Beck and Yue Gong. 2013. Wheel-spinning: Students who fail to master a skill. In *Int. conf. on artificial intelligence in education*. Springer, 431–440.

[5] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proc. of the working group reports on innovation and technology in computer science education* (2019).

[6] Neil CC Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proc. of the tenth annual Conf. on Int. computing education research*. 43–50.

[7] Neil CC Brown and Amjad Altadmri. 2017. Novice Java programming mistakes: Large-scale data vs. educator beliefs. *ACM Trans. on Computing Education (TOCE)* 17, 2 (2017), 1–21.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. In *Proc. of the 54th ACM Technical Symp. on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. ACM, New York, NY, USA, 1136–1142. https://doi.org/10.1145/3545945.3569823

[10] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A Becker, and Brent N Reeves. 2023. Promptly: Using Prompt Problems to Teach Learners How to Effectively Utilize AI Code Generators. *arXiv preprint arXiv:2307.16364* (2023).

[11] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proc. of the 17th ACM annual Conf. on Innovation and technology in computer science education*. 75–80.

[12] Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. 2023. Computing Education in the Era of Generative AI. *arXiv preprint arXiv:2306.02608* (2023).

[13] Paul Denny, James Prather, Brett A Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B Powell. 2021. On Designing Programming Error Messages for Novices: Readability and Its Constituent Factors. In *Proc. of the 2021 CHI Conf. on Human Factors in Computing Systems*. 1–15.

[14] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common Logic Errors Made by Novice Programmers. In *Proc. of the 20th Australasian Computing Education Conf.* ACM, New York, NY, USA, 83–89.

[15] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th Int. Conf. on Software Engineering (ICSE)*. IEEE, 1469–1481.

[16] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conf.* (Virtual Event, Australia) *(ACE '22)*. ACM, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[17] Sue Fitzgerald, Gary Lewandowski, Renee McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116.

[18] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Trans. on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–35.

[19] Ava Heinonen, Bettina Lehtelä, Arto Hellas, and Fabian Fagerholm. 2023. Synthesizing research on programmers' mental models of programs, tasks and concepts—A systematic literature review. *Information and Software Technology* (2023), 107300.

[20] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. *arXiv preprint arXiv:2306.05715* (2023).

[21] Irene Hou, Owen Man, Sophie Mettille, Sebastian Gutierrez, Kenneth Angelikas, and Stephen MacNeil. 2023. More Robots are Coming: Large Multimodal Models (ChatGPT) can Solve Visually Diverse Images of Parsons Problems. *arXiv preprint arXiv:2311.04926* (2023). https://doi.org/10.48550/arXiv.2311.04926

[22] Irene Hou, Sophia Mettille, Owen Man, Zhuo Li, Cynthia Zastudil, and Stephen MacNeil. 2024. The Effects of Generative AI on Introductory Students' Help-Seeking Preferences. In *Australasian Computing Education Conference (ACM ACE '24)*.

[23] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020* (2023).

[24] W. Lewis Johnson, Elliot Soloway, Benjamin Cutler, and Steven Draper. 1983. *Bug Catalogue: I.* Technical Report. Yale University, YaleU/CSD/RR #286.

[25] Samia Kabir, David N Udo-Imeh, Bonan Kou, and Tianyi Zhang. 2023. Who Answers It Better? An In-Depth Analysis of ChatGPT and Stack Overflow Answers to Software Engineering Questions. *arXiv preprint arXiv:2308.02312* (2023).

[26] Tobias Kohn. 2019. The error behind the message: Finding the cause of error messages in python. In *Proc. of the 50th ACM Technical Symp. on Computer Science Education*. 524–530.

[27] Teemu Koivisto and Arto Hellas. 2022. Evaluating CodeClusters for Effectively Providing Feedback on Code Submissions. In *2022 IEEE Frontiers in Education Conf. (FIE)*. IEEE, 1–9.

[28] Charles Koutcheme, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. 2023. Automated Program Repair Using Generative Models for Code Infilling. In *Int. Conf. on Artificial Intelligence in Education*. Springer, 798–803.

[29] Sam Lau and Philip J. Guo. 2023. From 'Ban It Till We Understand It' to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot. In *In Proc. of the 2023 ACM Conf. on Int. Computing Education Research V.1 (ICER '23 V1)*. ACM. https://doi.org/10.1145/3568813.3600138

[30] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing Code Explanations Created by Students and Large Language Models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 124–130. https://doi.org/10.1145/3587102.3588785

[31] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using large language models to enhance programming error messages. In *Proc. of the 54th ACM Technical Symp. on Computer Science Education V. 1*. 563–569.

[32] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4 (2004), 119–150.

[33] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further Evidence of a Relationship between Explaining, Tracing and Writing Skills in Introductory Programming. *SIGCSE Bull.* 41, 3 (2009), 161–165.

[34] Stephen MacNeil, Joanne Kim, Juho Leinonen, Paul Denny, Seth Bernstein, Brett A. Becker, Michel Wermelinger, Arto Hellas, Andrew Tran, Sami Sarsa, James Prather, and Viraj Kumar. 2023. The Implications of Large Language Models for CS Teachers and Students. In *Proc. of the 54th ACM Technical Symp. on Computer Science Education V. 2*. ACM, New York, NY, USA, 1255.

[35] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development

E-Book. In *Proc. SIGCSE'23*. ACM, 6 pages.
[36] Stephen MacNeil, Andrew Tran, Juho Leinonen, Paul Denny, Joanne Kim, Arto Hellas, Seth Bernstein, and Sami Sarsa. 2023. Automatically Generating CS Learning Materials with Large Language Models. In *Proc. of the 54th ACM Technical Symp. on Computer Science Education V. 2*. ACM, New York, NY, USA, 1176.
[37] Yana Malysheva and Caitlin Kelleher. 2020. Bugs as Features: Describing Patterns in Student Code through a Classification of Bugs. In *Extended Abstracts of the 2020 CHI Conf. on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–7.
[38] Davin McCall and Michael Kölling. 2019. A new look at novice programmer errors. *ACM Trans. on Computing Education (TOCE)* 19, 4 (2019), 1–30.
[39] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*. 125–180.
[40] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *ACM SIGCSE Bulletin*, Vol. 40. ACM.
[41] James Prather, Paul Denny, Juho Leinonen, Brett A Becker, Ibrahim Albluwi, Michael E Caspersen, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, et al. 2023. Transformed by Transformers: Navigating the AI Coding Revolution for Computing Education: An ITiCSE Working Group Conducted by Humans. In *Proc. of the 2023 Conf. on Innovation and Technology in Computer Science Education V. 2*. 561–562.
[42] James Prather, Paul Denny, Juho Leinonen, Brett A Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, et al. 2023. The robots are here: Navigating the generative ai revolution in computing education. *arXiv preprint arXiv:2310.00658* (2023).
[43] Ben Puryear and Gina Sprint. 2022. Github copilot in the classroom: learning to code with AI assistance. *J. of Computing Sciences in Colleges* 38, 1 (2022), 37–47.
[44] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proc. of the 2022 ACM Conf. on Int. Computing Education Research - Volume 1*. ACM, 27–43.
[45] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. 2023. Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses. *The 19th ACM Conference on International Computing Education Research (ICER)* (2023).
[46] Jaromir Savelka, Arav Agarwal, Christopher Bogart, and Majd Sakr. 2023. Large Language Models (GPT) Struggle to Answer Multiple-Choice Questions about Code. arXiv:2303.08033 [cs.CL]
[47] Jaromir Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. 2023. Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses? *arXiv preprint arXiv:2303.09325* (2023).
[48] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do we know how difficult the rainfall problem is?. In *Proc. of the 15th Koli Calling Conf. on Computing Education Research*. 87–96.
[49] Chenglei Si, Zhe Gan, Zhengyuan Yang, Shuohang Wang, Jianfeng Wang, Jordan Boyd-Graber, and Lijuan Wang. 2023. Prompting GPT-3 To Be Reliable. arXiv:2210.09150 [cs.CL]
[50] Rebecca Smith and Scott Rixner. 2019. The error landscape: Characterizing the mistakes of novice programmers. In *Proc. of the 50th ACM technical Symp. on computer science education*. 538–544.
[51] Elliot Soloway, Jeffrey G. Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM* 26, 11 (1983), 853–860.
[52] Elliot Soloway, Kate Ehrlich, Jeffrey G. Bonar, and Judith Greenspan. 1982. What do novices know about programming? In *Directions in Human–Computer Interactions*. Vol. 6. Ablex Publishing, 27–54.
[53] James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.
[54] Sean Teebagy, Lauren Colwell, Emma Wood, Antonio Yaghy, and Misha Faustina. 2023. Improved performance of ChatGPT-4 on the OKAP exam: A comparative study with ChatGPT-3.5. *medRxiv* (2023), 2023–04.
[55] Andrew Tran, Linxuan Li, Egi Rama, Kenneth Angelikas, and Stephen MacNeil. 2023. Using Large Language Models to Automatically Identify Programming Concepts in Code Snippets. In *Proc. of the 2023 ACM Conf. on Int. Computing Education Research - Volume 2*, Vol. 1. ACM, 563–569.
[56] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin* 39, 3 (2007), 236–240.
[57] Michel Wermelinger. 2023. Using GitHub Copilot to Solve Simple Programming Problems. In *Proc. of the 54th ACM Technical Symp. on Computer Science Education V. 1*. ACM.
[58] Jacqueline Whalley, Tony Clear, and RF Lister. 2007. The many ways of the BRACElet project. *Bull. of Applied Computing and Information Technology* (2007).
[59] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Novice reflections on debugging. In *Proc. of the 52nd ACM Technical Symp. on Computer Science Education*. 73–79.
[60] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proc. of the 8th Australasian Conf. on Computing Education - Volume 52*. Australian Computer Society, Inc., AUS, 243–252.
[61] Michael Winikoff. 2014. Novice programmers' faults & failures in GOAL programs. In *Proc. of the 2014 Int. Conf. on Autonomous agents and multi-agent systems*. 301–308.
[62] Cynthia Zastudil, Magdalena Rogalska, Christine Kapp, Jennifer Vaughn, and Stephen MacNeil. 2023. Generative AI in Computing Education: Perspectives of Students and Instructors. *arXiv preprint arXiv:2308.04309* (2023). https://doi.org/10.48550/arXiv.2308.04309
[63] Tony Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-Shot Performance of Language Models. In *Int. Conf. on Machine Learning*.