Kerslake, Chris; Denny, Paul; Smith, David H.; Leinonen, Juho; MacNeil, Stephen; Luxton-Reilly, Andrew; Becker, Brett A.

Exploring Student Reactions to LLM-Generated Feedback on Explain in Plain English Problems

Published in:
SIGCSE TS 2025 - Proceedings of the 56th ACM Technical Symposium on Computer Science Education

DOI:
10.1145/3641554.3701934

Published: 18/02/2025

Document Version
Publisher's PDF, also known as Version of record

Published under the following license:
CC BY

Please cite the original version:
Kerslake, C., Denny, P., Smith, D. H., Leinonen, J., MacNeil, S., Luxton-Reilly, A., & Becker, B. A. (2025). Exploring Student Reactions to LLM-Generated Feedback on Explain in Plain English Problems. In *SIGCSE TS 2025 - Proceedings of the 56th ACM Technical Symposium on Computer Science Education* (Vol. 1, pp. 575-581). ACM. https://doi.org/10.1145/3641554.3701934

Exploring Student Reactions to LLM-Generated Feedback on Explain in Plain English Problems

Chris Kerslake
Simon Fraser University
Burnaby, BC, Canada
chris.kerslake@sfu.ca

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

David H. Smith IV
University of Illinois
Urbana, IL, USA
dhsmith2@illinois.edu

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

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

Andrew Luxton-Reilly
University of Auckland
Auckland, New Zealand
a.luxton-reilly@auckland.ac.nz

Brett A. Becker
University College Dublin
Dublin, Ireland
brett.becker@ucd.ie

Abstract

Code reading and comprehension skills are essential for novices learning programming, and explain-in-plain-English tasks (EiPE) are a well-established approach for assessing these skills. However, manual grading of EiPE tasks is time-consuming and this has limited their use in practice. To address this, we explore an approach where students explain code samples to a large language model (LLM) which generates code based on their explanations. This generated code is then evaluated using test suites, and shown to students along with the test results. We are interested in understanding how automated formative feedback from an LLM guides students' subsequent prompts towards solving EiPE tasks. We analyzed 177 unique attempts on four EiPE exercises from 21 students, looking at what kinds of mistakes they made and how they fixed them. We found that when students made mistakes, they identified and corrected them using either a combination of the LLM-generated code and test case results, or they switched from describing the purpose of the code to describing the sample code line-by-line until the LLM-generated code exactly matched the obfuscated sample code. Our findings suggest both optimism and caution with the use of LLMs for unmonitored formative feedback. We identified false positive and negative cases, helpful variable naming, and clues of direct code recitation by students. For most students, this approach represents an efficient way to demonstrate and assess their code comprehension skills. However, we also found evidence of misconceptions being reinforced, suggesting the need for further work to identify and guide students more effectively.

CCS Concepts

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

SIGCSE TS 2025, February 26-March 1, 2025, Pittsburgh, PA, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0531-1/25/02
https://doi.org/10.1145/3641554.3701934

Keywords

formative feedback; misconceptions; explain in plain English; EiPE; large language models; LLM; qualitative analysis

ACM Reference Format:
Chris Kerslake, Paul Denny, David H. Smith IV, Juho Leinonen, Stephen MacNeil, Andrew Luxton-Reilly, and Brett A. Becker. 2025. Exploring Student Reactions to LLM-Generated Feedback on Explain in Plain English Problems. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025), February 26-March 1, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3641554.3701934

1 Introduction

Learning to read and comprehend code has long been an important skill for novice programmers [16, 36]. In recent years, as generative AI technologies such as large language models (LLMs) have become capable of generating code automatically, the ability to understand code is becoming increasingly important [7]. Traditional methods for building and assessing code comprehension often involve "Explain in Plain English" (EiPE) tasks, where students are asked to describe the functionality of a piece of code. These tasks help students develop a deeper understanding of code semantics and logic, but they are challenging to grade objectively and at scale [9, 14].

Recent advances in LLMs offer new approaches for generating automated feedback on EiPE tasks [8, 25, 26]. In these methods, a student's explanation of a code snippet is used to generate new code, which is then evaluated against a pre-written test suite to check for functional equivalence with the original code. The feedback provided to students typically includes the LLM-generated code and the results of the test cases, indicating which tests passed and which failed. This automated process can provide immediate feedback to students, potentially enhancing their learning experience by allowing them to identify and correct their mistakes in real-time.

Prior work in this area has primarily focused on quantitative analyses that examine overall success rates of student attempts to solve EiPE questions [8, 26]. These studies have demonstrated the potential benefits of LLM-based feedback mechanisms but have not thoroughly explored the quality or accuracy of the feedback provided to students. One critical aspect is the occurrence of false positives and false negatives. A false positive occurs when a student's incorrect prompt results in correct code generated by the LLM, while a false negative occurs when a student's correct prompt results in incorrect code generated by the LLM. Understanding these error rates is crucial for evaluating the reliability of LLM-generated feedback, as false positives in particular appear to be measurably harmful to learning [15]. Thus, there is a need for additional qualitative analyses to ensure that LLM-generated feedback is both accurate and helpful, and to identify areas where it can be improved.

In this work, we qualitatively analyse student attempts at solving EiPE tasks and the corresponding feedback generated by an LLM. We examine how students respond to this feedback, particularly when their initial attempts are incorrect. Our over-arching goal is to understand how automated formative feedback from an LLM influences students' subsequent attempts and to identify ways to enhance the feedback to provide more accurate and useful guidance. We address the following questions:

- **RQ1**: What rates are observed for false positives and false negatives in LLM-generated feedback on EiPE tasks?
- **RQ2**: How do students react to LLM-generated feedback when solving EiPE tasks?
- **RQ3**: When students make mistakes, what feedback do they use to correct them?

2 Related Work

The skills required to program are often separated into distinct categories for the purposes of teaching and evaluation [11, 36]. Among these are: code writing [34], tracing [12], and comprehension [1, 35]. There is evidence that code tracing and code comprehension predict code writing [31] performance, which supports the notion that effective code writing subsumes tracing and comprehension. Before the advent of LLMs and their code generation capabilities, developing students' abilities to write code independently was seen as the primary goal of introductory programming courses. However, as we enter the age of AI-assisted programming an emerging narrative within computer science education is that the priority should shift from teaching students to code *independently* towards instructing them to program with the assistance of LLMs [20, 30] with some going as far as to suggest we should adopt a "Prompts First" approach [22].

A commentary on *CS2023: ACM/IEEE-CS/AAAI Computer Science Curricula* suggests that the skill of comprehending code should be emphasized because students will be required to spend substantial amounts of time evaluating the functionality of AI-generated code [2]. Code comprehension is frequently assessed by asking students to describe a code segment using natural language via EiPE questions [16]. Though grading standards for these questions vary [5, 32], ideal responses are often characterized by their ability to *correctly* describe the code's *purpose* rather than its implementation [10]. To characterize student understanding, the SOLO taxonomy (Structure of Observed Learning Outcomes [3]) is typically deployed, where a description of the code implementation is aligned with the *multi-structural* level of SOLO and a description of the purpose of the code is aligned with the *relational* level [23].

Despite the clear utility of EiPE questions, the adoption of these questions at scale has been limited by the difficulty of automatically grading natural language responses. Prior to the widespread availability of large language models, the only automatic grading approach developed and evaluated for EiPE questions was introduced by Fowler et al. [9]. This approach utilizes a logistic classifier trained on a large quantity (500-600) of human labeled EiPE responses. Though this approach achieves results similar to that of a trained teaching assistant it suffers from several core limitations: 1) the overhead of human labeling needed for question authoring, 2) the inability to provide feedback beyond dichotomous, 3) the limited transparency of the grading mechanism for students [14].

A more recent LLM-based approach for grading EiPE questions introduced by Smith IV and Zilles [27] involves generating code from a student's EiPE response and grading the generated code via unit tests to identify if the generated code is functionally equivalent to the code the student was describing. This approach addresses each of the limitations of the logistic classifier approach by: 1) reducing the question authoring process to simply writing test cases, 2) providing students feedback through test cases and the generated code, and 3) increasing the transparency of the standards by which the students are being graded. An evaluation of these questions in lab activities by Denny et al. [8] found that *relational* responses were more successful than *multistructural* responses suggesting something of an alignment between what makes effective prompts and ideal EiPE responses. Follow-up work by Smith IV et al. [26] found that students were largely positive and engaged with the feedback in two modes: (i) comparing the generated code to the code they were describing to identify differences; and (ii) observing test case output to identify edge cases they had missed in their prompt. Many students explicitly stated that the feedback contributed to their ability to comprehend the code and form successful responses.

3 Methods

The EiPE tasks used in this study were delivered to students during the eighth week of an introduction to programming course at the University of Auckland, a large research university in New Zealand. The twelve-week course was split into two six-week modules that covered MATLAB and the C programming language. The first module focused on typical CS1 topics, including variables, arrays, conditionals, loops, functions, and common algorithms in MATLAB. The second module revisited the same concepts using the C programming language.

Following approval by the university's human ethics committee, data was collected from 861 students enrolled in the course. The participants were first-year engineering students and were not expected to have any formal programming experience

Table 1: List of the four EiPE questions and their descriptions.

Q#	Task	Description
Q1	FindSumBetween	calculate sum between two values
Q2	CountEvensInArray	count num. even values in an array
Q3	LastZeroInArray	find index of last zero in an array
Q4	SumPositiveValues	sum the positive values in an array

3.1 Explain in Plain English (EiPE) Questions

During their lab for the eighth week of the course, students were tasked to answer four code-related questions (Table 1) using the online assessment platform PrairieLearn [33]. Following examples from previous EiPE studies [10, 18], for each of the four questions, students were presented with a single C function (named 'foo' with deliberately obfuscated variable names) and asked to describe the function in plain English. After submitting their answer, a prompt to generate a solution was created and submitted to GPT-3.5. The resulting C code was then evaluated against a set of test cases, and both the code and the results from the test cases were shown to the student. If the LLM-generated code successfully passed all test cases, then the code was considered functionally equivalent, and the question was marked as passed.

3.2 Student Post-Lab Reflections

After completing all four of the EiPE questions, students were asked to answer a set of feedback questions. One of the questions, Question 10 asked students to describe their approach when their initial response was unsuccessful. To evaluate these post-lab reflections, the first author generated a set of inductive codes based on their interpretation of how each student responded (see Table 4).

3.3 Sampling Procedures & Inter-Rater Reliability

For this study, a subset of 21 students were randomly selected from the larger group of 861 students in three rounds of sampling. In the first round, ten students were randomly selected, and cases were created for each student that covered their prompts, code, and test results for each of the four lab questions, as well as their post-lab feedback comments. After the initial ten cases were inductively coded, two additional groups of students were targeted to broaden the set of codes. Six students were randomly chosen from those who had made at least one error, and five more students were chosen from those who had made at least 11 attempts during their exercises. In total, 177 prompts from 21 students were examined.

To validate the inductively generated codes by the first author, cases were added until saturation was reached. After reaching saturation, a second coder coded a subset of the data until a Cohen's kappa of .80 (strong agreement) was achieved.

3.4 False Positives & False Negatives

To evaluate student descriptions for false positives and false negatives, an expert programmer read each student prompt to determine what the prompt described. In most cases, the LLM-generated code followed logically from the description provided. However, in some cases, the description did not provide sufficient or correct information but passed (false positive), or the description provided was deemed sufficient but did not pass (false negative).

3.5 Student Reactions to LLM Feedback

To evaluate how students reacted to LLM-generated feedback when solving EiPE tasks, the first author generated an inductive set of codes based on their interpretation of what each prompt described.

4 Findings

4.1 False Positives & False Negatives

Of the 177 total prompts that were analyzed, 2 false positives were identified, and 5 false negatives were identified (3.95% of prompts). However, these false identifications were experienced by 5 of the 21 students analyzed (23.8%), and although the sample size is small, the occurrence was greater than expected.

4.1.1 False Positives.

False positives occur when the LLM generates code that satisfies the tests based on an incorrect problem description. False positives have the effect of 'passing' a student's description when it should not have passed. As a result, a false positive could accidentally reinforce an incorrect idea or concept. We discovered two false positives: a vague answer resulted in a pass, and the same prompt was presented twice, resulting in two different answers, one of which passed.

In the first case, Student 103 provided a vague answer to Q3 (LastZeroInArray) for their second prompt (Prompt-103-Q3-02). This resulted in code that passed all of the test cases. This was not expected, and similar prompts that lacked explicit instruction on what to return resulted in 'failed' rather than 'success' results with similar prompts.

> "takes an integer array and its length and gain an integer of the index of the last occurrence of 0"

Prompt-103-Q3-02

Another type of false positive that was encountered resulted from the non-deterministic nature of LLMs. The LLM generated different answers for the same prompt in this second type. For example, Student 112 provided a vague recitation of the sample code in response to Q1 (FindSumBetween) (Prompt-112-Q1-07). As expected, the resulting code (Code-112-Q1-07) failed two of the four test cases. However, when they resubmitted the same prompt again (Prompt-112-Q1-08), the generated code was completely different (Code-112-Q1-08) and passed all of the test cases.

> "takes an integer a and b and returns an updated x value by adding the i value to it"

Prompt-112-Q1-07 & Prompt-112-Q1-08

```c
int foo(int a, int b) {
    int x = a + b;
    return x;
}
```

Source Code 1: Code-112-Q1-07

```c
int foo(int a, int b) {
    int x = 0;
    for (int i = a; i <= b; i++) {
        x += i;
    }
    return x;
}
```

Source Code 2: Code-112-Q1-08

4.1.2 False Negatives.

Unlike false positives, where students pass when they should not, false negatives occur when the LLM fails to generate code for a seemingly valid prompt. In these cases, the student is misled that their prompt is invalid, and thus, they could spend time looking for errors that do not exist, or it could cause them to question their understanding unnecessarily. We discovered two types of false negatives, one where the LLM failed to consistently include a function parameter (thus breaking the test cases), and one where the LLM was uncharacteristically literal in its prompt interpretation.

For the first type of false negative, Student 102 described only the first input parameter as an integer array but neglected to mention its length as the second parameter. This was a common pattern, and in the majority of cases, this omission had no ill effects. However, in the case of 102's answer to Q2 (CountEvensInArray), their prompt (Prompt-102-Q2-01) resulted in the LLM omitting the unstated length parameter , which resulted in four failed test cases. At first, it seems reasonable that the LLM would not include the second parameter. However, the hidden system prompt used by the EiPE tool explicitly instructs the model to include them and when other students failed to mention any of the input parameters, the LLM automatically included both parameters. In 102's case, the student identified that the LLM had failed to include the second parameter and adjusted their subsequent prompts to describe each parameter explicitly. As a result, rather than this student moving towards an abstract description with no function signature details, they instead began to over-fit their description to the provided sample code.

"takes an integer array and returns the amount of even numbers there are in the array"

Prompt-102-Q2-01

"takes an integer array *and the length of the array*, and then returns the amount of even numbers there are in the array"

Prompt-102-Q2-02

For the second type of false negative, Student 111 provided a reasonable answer for Q2 (CountEvensInArray) (Prompt-111-Q2-02) that resulted in a literal interpretation of the answer where each item in the array was first divided by two and then their new value was checked if it was divisible by two. Again, the student compared the generated code, noticed the literal interpretation, and correctly adjusted their prompt (Prompt-111-Q2-03). Although traditional programming requires precision and specificity, in this case, the LLM's literal interpretation of the prompt was not consistent with the goal of providing a high-level explanation rather than a recitation of the code.

"takes an array of integers and divides each number in the array by 2 and checks how many numbers have a remainder of 0"

Prompt-111-Q2-02

"takes an array of integers and determines how many are divisible by 2 with no remainder"

Prompt-111-Q2-03

Table 2: Student 111 Q3 Prompts. F - Failed; S - Success

Tries	F/S	Prompt
Q3-01	F	takes an array of integers and determines how many values are equal to 0
Q3-02	F	takes an array of integers and determines which numbers are equal to 0
Q3-03	F	takes an array of integers, starting with a counter at -1 and checks to see if the values are 0. If so then it will assign the counter to the position of the value that is equal to zero.
Q3-04	F	takes an array of integers, starting with a counter at -1 and checks to see if the values are 0. If so, then it will assign the counter to the position of the value that is equal to zero, *returning the count.*
Q3-05	S	takes an array of integers and outputs an integer that represents the position of the last zero in the array, if there are no zeros output -1

4.2 Student Reactions to LLM Feedback

Students predominately exhibited one of two approaches to explaining the code samples, either at an abstract high-level or by directly reciting the code in detail to reverse engineer the prompt from the code. This section explores how two students reacted to feedback when their initial prompt failed.

As an example of a student who provided predominately high-level descriptions, Student 111 (shown in Table 2) started off with an incorrect description (Q3-01) for Q3 (LastZeroInArray), misidentifying the code as counting how many zeros rather than finding the index of the last zero in the array. As a result, their first prompt refers to the number of zeros. This results in the LLM generating code that counts the zeros and a variable named "count" (Code-111-Q3-01). As expected, all four test cases failed.

```
int foo(int arr[], int size) {
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (arr[i] == 0) {
            count++;
        }
    }
    return count;
}
```

Source Code 3: Code-111-Q3-01

In their post-lab feedback, Student 111 stated that they "didn't really use the feedback given by the system." Instead, they "just looked at the code it outputted and what I needed to do to fix that code." This is supported by their next prompt (Q3-02) where they switch from counting with "how many" into "which" numbers are equal to 0. Again, their second prompt also fails, but this time in a more dramatic fashion by first breaking the function type declaration from int to void and not returning "which numbers are equal to 0" but instead printing them to the screen.

In response, the student switches from a high-level description to describing lines from the sample code in an effort to get the LLM to generate the same code (Q3-03). Of particular interest is that rather than referring to the variable "x" from the sample code, they use the name “counter” from their description and the originally generated code (Code-111-Q3-01). Also, after the function definition switched from int to void, the student switched to explicitly describing at least the input array for the function. Note that they did not also include the second function parameter “size”, but the LLM included it anyway (in contrast to the false negative results discussed in the previous section). The resulting code for their third attempt incorrectly determined the *first* position of a zero in the array and also accidentally passed the first test case.

For their fourth attempt (Q3-04), the student explicitly added a return statement (again, not required by most other prompts), but the LLM unexpectedly dropped the implicit second parameter of “size” resulting in a compilation error due to a function definition mismatch and all tests failing.

We had expected them to add the second function parameter for their fifth (and final) attempt (Q3-05), but they did not. However, they determined that the code was looking for the *last* zero rather than counting zeros, so they changed their prompt to describe that. As a result, the LLM generated the correct function signature, and the code passed all four test cases.

In contrast to the high-level descriptions provided by Student 111, Student 128 directly recited the sample code for their explanations. As a result, they reacted more to the generated code and adjusted their prompts in response to specific mismatched lines of code, as shown in Table 3.

Table 3: Student 128 Q1 Prompts. F - Failed; S - Success

Tries	F/S	Prompt
Q1-01	F	Passes two inputs which are both integers named ’a’ and ’b’. Return the amount of values that are less than or equal to ’b’ but greater than or equal to ’a’.
Q1-03	F	Passes two inputs which are both integers named ’a’ and ’b’. An integer called ’x’ is created which is equal to 0. *In a for loop where ’i’ equals to ’a’ add 1 to ’x’ if the current value of i is less than or equal to ’b’. if it is less than or equal to then add 1 to ’i’ and repeat the test.* Return ’x’.
Q1-05	S	Passes two inputs which are both integers named ’a’ and ’b’. An integer called ’x’ is created which is equal to 0. In a for loop where ’i’ equals to ’a’ *add the value of ’i’ to ’x’* if the current value of ’i’ is less than or equal to ’b’. If it is less than or equal to then add 1 to ’i’ and repeat the test. Return ’x’.

4.3 Student Post-Lab Reflections

At the end of the lab, students were asked to describe what information or techniques they used to fix their failed prompts.

For students who provided the correct answer the first time or succeeded on all of the tests, they reported not using any of the feedback. The rest of the students, as shown in Table 4, reported using both the generated code and the failed test cases to guide their subsequent changes after failing. Thus, the generated code did help guide them, and the addition of the test cases was beneficial.

Table 4: Answers to the post-lab feedback survey. “What information or technique(s) do they report using to fix failed prompts?” (N=21 students)

Code	Information Source	Reported %
READ	Re-read own explanation for mistakes.	13.3%
TESTS	Compared against test cases.	13.3%
CODE	Compared against generated code.	26.7%
BOTH	Used both CODE & TESTS.	46.7%

Tables 5 & 6 report specific student quotes about each of the two feedback sources.

Table 5: Feedback from students who used generated code

S#	Quote
14	*I would look at the code output and then re-adjust my answer.*
100	*I tried to use the code generated to identify what keywords within my prompt were causing it to generate code in that specific way and then tried to isolate those words and change them in order to fit the intended purpose within my prompt.*
140	*My approach was to look at the code that I got and the original code and compare how similar they were (or how they could work the same if they did not look similar).*

Table 6: Feedback from students who used test cases

S#	Quote
12	*I just checked each component of my request after looking at what test cases failed until I found the problem.*
102	*I would check whether the test cases match what i was expecting from my response to fix it.*
150	*I used test cases as my main feedback only using generated code if I was failing multiple times because often my generated code and the code provided didn’t look the same but had the same functionality.*

As a counter-point to using the feedback productively, one student (510) noted that they got all of their questions correct on their first attempt, so they did not use the feedback. As a result, they did not look at the generated code. However, when looking at the student’s prompts, it was noted that they used a combination of over-explaining and high-level answering. Although the student did provide a sufficient answer, they did not discover that their answer was over-complicated and longer than was necessary for the LLM. Thus, they were actually less efficient, prompt-wise, than they needed to be. Their explanations were still well articulated and seemed aimed at other programmers more so than an LLM, but from an LLM perspective, they were unnecessarily specific and complicated, resulting in the student not learning that they could be less verbose to achieve the same goal.

4.4 Unexpectedly Helpful Variable Names

Students who gave a valid description of the code at a high-level received additional feedback from the LLM in the form of useful variable names that reflected their description. For example, if they discussed counting something, then the return variable was often “count”, similar to summing or adding together. However, for students who directly recited the code, the variable names used by the LLM were those recited in the description. Hence, if they noted the variable x, the LLM used x, and the return value was x, the same as the sample code. This has two implications. First, this may be a way of detecting code recitation by parsing the variable names from the LLM-generated code. Second, this indicates that the student is probably not operating at a high-level and is only describing or seeing the syntax rather than the functionality of the sample code.

5 Discussion

The integration of generative AI into the workflows of computing professionals and students is having a big impact on coding and debugging. Effective use of these tools hinges on the ability to prompt AI models and interpret their responses accurately. Prior research indicates that the benefits of these tools are not uniformly distributed among students, largely due to differences in students' prompting and interpretation skills [13, 21].

Our study contributes to this discourse by offering insights into the challenges students encounter when prompting AI models, particularly focusing on the rates of false positives and false negatives in AI-generated feedback. False positives can give students unwarranted confidence in their abilities, while false negatives may mislead students into thinking they made a mistake, potentially leading students to question their abilities. However, false negatives may also allow students to further refine and better articulate their explanations, as prompts that do not reliably produce correct code can signify explanations that could be made more robust. This experience of dealing with non-deterministic outputs and debugging model responses is also arguably a competency that computing professionals will need in the future.

Although code reading tasks are typically designed with familiar code structures to build students' confidence, AI-generated code exposes students to novel approaches they can use to learn. Research suggests that it is beneficial [28, 29] to expose learners to ‘critical variations’ [17], which are carefully selected examples that highlight key differences in a phenomenon. Our findings show evidence of critical variations that can help students deepen their understanding of code structures, which is consistent with variation theory. However, because AI examples are generated probabilistically rather than selected intentionally, students' learning experiences may vary widely. Future work may consider ways to intentionally include variation in critical dimensions of generated code to leverage the benefits of being exposed to key variations. Future work may also wish to replicate our study in other natural language contexts, given the potential for LLM-powered pedagogies to extend beyond English [24].

We found evidence that the level of precision students use impacts the quality of the code generated, often by including meaningful variable names that helped reinforce the student's understanding. The precision required during prompting also helped expose when students were mixing up concepts with superficial similarities, such as loops used for counting versus summing.

Students used feedback to support reflection on their own explanations, test cases, and generated code. They tended to review the generated code or a combination of the code and tests most frequently to fix problems with their explanations. Based on this feedback, students were able to switch between prompting strategies, often from an initial high-level description, to other strategies, such as including code or writing pseudo code. This iterative process of revising prompts has been linked to improved outcomes [6, 19]. Additionally, an interview study highlighted that the willingness to iterate on prompts distinguishes those who successfully use generative AI from those who do not [13].

6 Limitations

We had students complete EiPE tasks to improve their ability to explain code. However, the task is not strictly an EiPE task. Prompt engineering techniques have the potential to affect generated code. For example, prior work has shown how including gratitude within prompts or offering bribes can improve the model's performance [4], but these prompts would not constitute better explanations. This paper shows that non-deterministic model responses also affect performance. So these are two aspects that affect performance, but neither relates to the quality of the explanation. Essentially, we have two reasons to say that prompting and explaining aren't exactly the same thing. While that is a limitation, previous evidence shows that high-performing prompts appear to correlate with aspects of high-quality explanations [8]. So while EiPE and prompting are synonymous, this paper further demonstrates some of the learning benefits associated with EiPE tasks. The feedback appears to encourage students to review their explanations, test cases, and resulting code; activities we hope students engage in when writing and explaining their code. Additionally, prior work investigating success rates with prompting has shown that initial success rates are typically lower than eventual success rates [19].

7 Conclusion

Understanding code is becoming increasingly important with the advent of generative AI, because programmers are increasingly encountering and dealing with AI-generated code. We analyzed how students utilized feedback for Explain in Plain English tasks that were auto-graded with the help of LLMs. We found that although LLM-generated code did result in false positive and false negative answers, they were relatively rare. Also, while there were rare cases where misconceptions might have been reinforced due to false feedback, the vast majority of students were able to effectively use feedback from the generated code and test cases to reach the correct solution. These findings describe some current challenges but also provide insights for using LLMs to support student learning effectively in computing courses.

Acknowledgments

This research was partially supported by the Research Council of Finland (Academy Research Fellow grant number 356114).

References

[1] Sushmita Azad. 2020. *Lessons learnt developing and deploying grading mechanisms for EiPE code-reading questions in CS1 classes.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[2] Brett A Becker, Michelle Craig, Paul Denny, Hieke Keuning, Natalie Kiesler, Juho Leinonen, Andrew Luxton-Reilly, Lauri Malmi, James Prather, and Keith Quille. 2023. Generative AI in Introductory Programming. https://csed.acm.org/large-language-models-in-introductory-programming/

[3] John B Biggs and Kevin F Collis. 2014. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome).* Academic Press, New York, NY, USA.

[4] Sondos Mahmoud Bsharat, Aidar Myrzakhan, and Zhiqiang Shen. 2024. Principled Instructions Are All You Need for Questioning LLaMA-1/2, GPT-3.5/4. arXiv:2312.16171 [cs.CL] https://arxiv.org/abs/2312.16171

[5] Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, and Craig Zilles. 2020. A Validated Scoring Rubric for Explain-in-Plain-English Questions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20).* ACM, New York, NY, USA, 563–569. https://doi.org/10.1145/3328778.3366879

[6] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proc. of the 55th ACM Tech. Sym. on CS Ed V. 1 (SIGCSE 2024).* ACM, New York, NY, USA, 296–302. https://doi.org/10.1145/3626252.3630909

[7] 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. 2024. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan 2024), 56–67. https://doi.org/10.1145/3624720

[8] Paul Denny, David H. Smith, Max Fowler, James Prather, Brett A. Becker, and Juho Leinonen. 2024. Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024).* ACM, New York, NY, USA, 283–289. https://doi.org/10.1145/3649217.3653587

[9] Max Fowler, Binglin Chen, Sushmita Azad, Matthew West, and Craig Zilles. 2021. Autograding "Explain in Plain English" questions using NLP. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21).* ACM, New York, NY, USA, 1163–1169. https://doi.org/10.1145/3408877.3432539

[10] Max Fowler, Binglin Chen, and Craig Zilles. 2021. How should we 'Explain in plain English'? Voices from the Community. In *Proc. of the 17th ACM Conf. on Int. Computing Education Research.* ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/3446871.3469738

[11] Max Fowler, David H. Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. 2022. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. *Computer Science Education* 32, 3 (July 2022), 355–383. https://doi.org/10.1080/08993408.2022.2079866

[12] Matthew Hertz and Maria Jump. 2013. Trace-based teaching in early programming courses. In *Proc. of the 44th ACM Technical Symp. on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13).* ACM, New York, NY, USA, 561–566. https://doi.org/10.1145/2445196.2445364

[13] Irene Hou, Sophia Mettille, Owen Man, Zhuo Li, Cynthia Zastudil, and Stephen MacNeil. 2024. The Effects of Generative AI on Computing Students' Help-Seeking Preferences. In *Proc. of the 26th Australasian Computing Education Conf. (ACE '24).* ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/3636243.3636248

[14] Silas Hsu, Tiffany Wenting Li, Zhilin Zhang, Max Fowler, Craig Zilles, and Karrie Karahalios. 2021. Attitudes Surrounding an Imperfect AI Autograder. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21).* ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/3411764.3445424

[15] Tiffany Wenting Li, Silas Hsu, Max Fowler, Zhilin Zhang, Craig Zilles, and Karrie Karahalios. 2023. Am I Wrong, or Is the Autograder Wrong? Effects of AI Grading Mistakes on Learning. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) *(ICER '23).* ACM, New York, NY, USA, 159–176. https://doi.org/10.1145/3568813.3600124

[16] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *SIGCSE Bull.* 38, 3 (jun 2006), 118–122. https://doi.org/10.1145/1140123.1140157

[17] Ference Marton and Shirley Booth. 2013. *Learning and awareness.* Routledge, New York.

[18] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in plain English' questions: implications for teaching. In *Proc. of the 43rd ACM Tech. Sym. on CS Ed* (Raleigh, NC, USA) *(SIGCSE '12).* ACM, New York, NY, USA, 385–390. https://doi.org/10.1145/2157136.2157249

[19] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24).* ACM, New York, NY, USA, 1–26. https://doi.org/10.1145/3613904.3642706

[20] Leo Porter and Daniel Zingaro. 2024. *Learn AI-assisted Python programming: with GitHub Copilot and ChatGPT* (first edition ed.). Manning Publications, Shelter Island, New York.

[21] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In *Proc. of the 2024 ACM Conf. on Int. Comp. Ed. Research - Volume 1* (Melbourne, VIC, Australia) *(ICER '24).* ACM, New York, NY, USA, 469–486. https://doi.org/10.1145/3632620.3671116

[22] Brent N. Reeves, James Prather, Paul Denny, Juho Leinonen, Stephen MacNeil, Brett A. Becker, and Andrew Luxton-Reilly. 2024. Prompts First, Finally. https://doi.org/10.48550/arXiv.2407.09231 arXiv:2407.09231 [cs].

[23] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline L. Whalley. 2008. Going SOLO to assess novice programmers. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education* (Madrid, Spain) *(ITiCSE '08).* ACM, New York, NY, USA, 209–213. https://doi.org/10.1145/1384271.1384328

[24] David H. Smith, Viraj Kumar, and Paul Denny. 2024. Explain in Plain Language Questions with Indic Languages: Drawbacks, Affordances, and Opportunities. arXiv:2409.20297 [cs.CY] https://arxiv.org/abs/2409.20297

[25] David H. Smith and Craig Zilles. 2024. Code Generation Based Grading: Evaluating an Auto-grading Mechanism for "Explain-in-Plain-English" Questions. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024).* ACM, New York, NY, USA, 171–177. https://doi.org/10.1145/3649217.3653582

[26] David H. Smith IV, Paul Denny, and Max Fowler. 2024. Prompting for Comprehension: Exploring the Intersection of Explain in Plain English Questions and Prompt Writing. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale* (Atlanta, GA, USA) *(L@S '24).* ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/3657604.3662039

[27] David H. Smith IV and Craig Zilles. 2023. Code Generation Based Grading: Evaluating an Auto-grading Mechanism for "Explain-in-Plain-English" Questions. https://doi.org/10.48550/arXiv.2311.14903 arXiv:2311.14903 [cs].

[28] Jarkko Suhonen, Janet Davies, Errol Thompson, and Kinshuk. 2007. Applications of variation theory in computing education. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88* (Koli National Park, Finland) *(Koli Calling '07).* Australian Computer Society, Inc., AUS, 217–220.

[29] Michael Thuné and Anna Eckerdal. 2009. Variation theory applied to students' conceptions of computer programming. *European Journal of Engineering Education* 34, 4 (2009), 339–347.

[30] Annapurna Vadaparty, Daniel Zingaro, David H. Smith IV, Mounika Padala, Christine Alvarado, Jamie Gorson Benario, and Leo Porter. 2024. CS1-LLM: Integrating LLMs into CS1 Instruction. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024).* ACM, New York, NY, USA, 297–303. https://doi.org/10.1145/3649217.3653584

[31] Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (Berkeley, CA, USA) *(ICER '09).* ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1584322.1584336

[32] Renske Weeda, Cruz Izu, Maria Kallia, and Erik Barendsen. 2020. Towards an Assessment Rubric for EiPE Tasks in Secondary Education: Identifying Quality Indicators and Descriptors. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research (Koli Calling '20).* ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3428029.3428031

[33] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. In *2015 ASEE Annual Conference & Exposition.* American Society for Engineering Education (ASEE), Seattle, WA, USA, 26.1238.1–26.1238.14. https://doi.org/10.18260/p.24575 ISSN: 2153-5965.

[34] Jacqueline Whalley and Nadia Kasto. 2014. A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (Uppsala, Sweden) *(ITiCSE '14).* ACM, New York, NY, USA, 279–284. https://doi.org/10.1145/2591708.2591762

[35] J. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. K. A. Kumar, and C. Prasad. 2006. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education*, Denise Tolhurst and Samuel Mann (Eds.), Vol. 52. Australian Computer Society, Inc., Hobart, Australia, 243–252. https://hdl.handle.net/10292/15405

[36] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235