ABSTRACT
In feedback generation for logical errors in programming assignments, large language model (LLM)-based methods have shown great promise. These methods ask the LLM to generate feedback given the problem statement and a student's (buggy) submission and have several issues. First, the generated feedback is often too direct in revealing the error in the submission and thus diminishes valuable learning opportunities for the students. Second, they do not consider the student’s learning context, i.e., their previous submissions, current knowledge, etc. Third, they are not layered since existing methods use a single, shared prompt for all student submissions. In this paper, we explore using LLMs to generate a “feedback-ladder”, i.e., multiple levels of feedback for the same problem-submission pair. We evaluate the quality of the generated feedback-ladder via a user study with students, educators, and researchers. We have observed diminishing effectiveness for higher-level feedback and higher-scoring submissions overall in the study. In practice, our method enables teachers to select an appropriate level of feedback to show to a student based on their personal learning context, or in a progressive manner to go more detailed if a higher-level feedback fails to correct the student’s error.\(^1\)

Keywords
Feedback, Large Language Models, Programming Assignments

1. INTRODUCTION
One of the primary ways humans learn is through feedback. In a class, where a teacher has to interact with numerous students, manually providing appropriate feedback for each student can be time-consuming. However, the effect of immediate, just-in-time [3] and personalized[4] feedback on student learning is hugely positive. The current practice of human feedback through teachers and teaching assistants can often be one-size-fits-all and not-on-time for students. Intelligent tutoring systems and online learning platforms can enable automatic feedback and scale up teachers’ efforts, especially in the setting of learning programming. Due to the highly structured nature of programming languages, there exists a body of work on feedback generation for students in programming tasks that has some significant success.

Automated feedback methods in programming scenarios are categorized into edit-based feedback, which focuses on code edits to fix bugs, and natural language-based feedback, providing explanations, or suggestions conversationally. Edit-based approaches primarily target syntax errors or provide suggestions for fixing buggy code [11, 16]. Recent advancements in large language models (LLMs) allow for the generation of fluent utterances around feedback, leveraging generative capabilities and knowledge of code. Approaches for automated feedback generation also consider the type of student errors, distinguishing between syntax and logical errors.

Automated feedback for logical errors in programming education has also been studied. [6, 5] generates test cases and Socratic questions based on the student submission. [14] systematically compares the performance of ChatGPT and GPT-4 to that of human tutors for a variety of scenarios in programming education including feedback generation. Their method produces a concise, single-sentence hint for the bug in the program, which is evaluated using human annotators. Similarly, [15] investigates the role of LLMs in providing human tutor-style programming hints to students. It uses a similar prompting strategy to the one in [14] to generate hints augmented with the failing test case and the fixed program, and then validated by GPT3.5. [12] conducts an experimental study using GPT3.5 to generate real-time feedback for the students and measure the effect of the feedback. All of these methods lack personalization of feedback, only generate feedback of one type, and fall short in quality evaluation experiments for the generated feedback.

In this paper, we explore the capability of GPT-4 in generating “feedback-ladder”, a multilevel feedback cascade for a single student-written buggy program. Different students need different types of help when solving problems [18, 9] and also at various stages of learning, students' help-seeking behavior changes [2]. From these findings, we expect that
generating feedback at multiple levels, forming a ladder, will be helpful to students. Feedback at a lower, coarse-grained level supports students conceptually and without giving away the solution – suitable for students with higher abilities. Feedback at a higher, fine-grained level supports students more specifically and directly points out their mistakes – suitable for students with lower abilities and prevents them from getting stuck. Therefore, teachers can use the feedback-ladder for personalized feedback by showing different levels to students with different needs, possibly by starting at a lower level and moving up if the student still struggles. Our key research question is: Can we automatically generate relevant and effective feedback-ladders to address students’ logical errors in introductory programming assignments?

1.1 Contributions
In this paper, we detail a method for automatically generating feedback-ladders for logical errors in programming assignments using GPT-4. First, we propose an LLM-based method that automatically generates feedback-ladders to address logical errors in programming assignments. Second, we conduct a user study that consists of annotators who are students, instructors, or researchers in programming education to evaluate the quality of the generated feedback-ladder. We observe that each level has a similar effectiveness score across different questions, though higher levels have lower scores than the lower levels. We also observe that it is harder to generate effective feedback-ladder for higher-scoring submissions than lower-scoring submissions.

2. METHODOLOGY
2.1 Feedback-ladder
We define feedback-ladder as a set of varying levels of feedback for a student-submitted, incorrect (including partially correct) program instance, \(P_s\). Each level in the ladder corresponds to how much information is contained in the hint shown to the student. With increasing levels of feedback in the ladder, the learning gain of the student diminishes [10]. The feedback levels are defined as:

**Level 0 (Yes/No):** The feedback only indicates whether \(P_s\) is correct or not. If a student can utilize this level of feedback, then they are going to learn the most.

**Level 1 (Test Case):** The feedback generates a test case \(C_{\text{fail}}\), that when given to \(P_s\) will give the wrong answer. It consists of the input for \(C_{\text{fail}}\), the expected correct answer according to the problem, and what \(P_s\) gives as the output. Utilizing this level teaches students debugging skills, which are hard to learn as beginners, and very conducive to their learning.

**Level 2 (Hint):** The feedback generates a high-level description of the logical error in the code that is responsible for the failure. This level of feedback should focus on the conceptual error that a student might have and must avoid any suggestion regarding editing the code. Feedback on this level has the best balance for a student to correct the mistake, as well as achieve learning gains.

**Level 3 (Location):** This level points out the lines at which the mistake occurred in the program. The feedback should refrain from explicitly mentioning the actual mistake or any edit suggestions. A student with sufficient programming knowledge should be able to correct the mistake with feedback on this level.

**Level 4 (Edit):** At this level, the feedback generates edits that turn the student-submitted buggy program, \(P_s\), into a correct program. The feedback should modify the existing structure of \(P_s\) and produce something as close as possible while showing only the edits instead of the whole corrected program. The learning gain for this level is minimal, though it can be helpful for students who are absolute beginners.

2.2 Generating Feedback-Ladder
We use the definitions of different levels in feedback-ladder in the prompts for GPT-4 to generate feedback in each level for a student-submitted buggy program. The prompt includes these definitions plus both the problem statement (\(Q\)) and the buggy program \(P_s\) itself. We use the openAI API to prompt GPT-4. Table 1 shows an example prompt to generate the feedback-ladder. Table 2 shows the generated feedback-ladder for the prompt in Table 1 by GPT-4.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0</td>
<td>The feedback only indicates whether (P_s) is correct or not.</td>
</tr>
<tr>
<td>Level 1</td>
<td>The feedback generates a test case (C_{\text{fail}}), that when given to (P_s) will give the wrong answer.</td>
</tr>
<tr>
<td>Level 2</td>
<td>The feedback generates a high-level explanation of why the code failed in the test case.</td>
</tr>
<tr>
<td>Level 3</td>
<td>The feedback generates a high-level suggestion about the location in the code where the mistake occurred.</td>
</tr>
<tr>
<td>Level 4</td>
<td>The feedback generates edits that turn the student-submitted buggy program, (P_s), into a correct program.</td>
</tr>
</tbody>
</table>

3. EXPERIMENTS
3.1 Dataset
We use the dataset from the 2nd CSEDMD Data Challenge, hereafter referred to as the CSEDMD dataset [17]. It is a college-level, publicly-available dataset with students’ actual code submissions. It contains 246 college students’ 46,825 full submissions on each of the 50 programming questions over an entire semester with rich textual information on problem statements and student code submissions and other relevant metadata such as the programming concepts involved in each question and all error messages returned by the compiler. From the full dataset, we only take submissions that do not have syntax errors, since our work
**Table 2: Corresponding response for the prompt in Table 1**

<table>
<thead>
<tr>
<th>Level 0: Incorrect</th>
<th>Level 1:</th>
<th>Level 2:</th>
<th>Level 3:</th>
<th>Level 4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect</td>
<td>The program fails in the following case.</td>
<td>The code fails because it wrongly handles the condition for the sum of a and b.</td>
<td>The code fails because it mistakenly checks if the sum of a and b is less than or equal to 10 and greater than or equal to 20. This condition will never be true as a number cannot be both less than or equal to 10 and greater than or equal to 20 at the same time.</td>
<td>Change the if condition from ( (a + b \leq 10 \land a + b \geq 20) ) to ( (a + b \geq 10 \land a + b &lt; 20) ).</td>
</tr>
</tbody>
</table>

**Table 3: Inter-rater agreement between annotators measured in Pearson correlation coefficient (PCC) values.**

<table>
<thead>
<tr>
<th>Annotator</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.00</td>
<td>0.05</td>
<td>0.11</td>
<td>0.43</td>
<td>0.50</td>
<td>0.14</td>
<td>0.67</td>
<td>0.91</td>
<td>0.18</td>
<td>0.47</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0.65</td>
<td>1.00</td>
<td>0.20</td>
<td>0.13</td>
<td>0.07</td>
<td>0.53</td>
<td>0.48</td>
<td>0.14</td>
<td>0.40</td>
<td>0.37</td>
<td>0.39</td>
</tr>
<tr>
<td>C</td>
<td>0.11</td>
<td>0.20</td>
<td>1.00</td>
<td>0.05</td>
<td>0.09</td>
<td>0.05</td>
<td>0.12</td>
<td>0.37</td>
<td>0.06</td>
<td>0.48</td>
<td>0.22</td>
</tr>
<tr>
<td>D</td>
<td>0.04</td>
<td>0.13</td>
<td>0.05</td>
<td>1.00</td>
<td>0.11</td>
<td>0.10</td>
<td>0.14</td>
<td>0.06</td>
<td>0.06</td>
<td>0.09</td>
<td>0.12</td>
</tr>
<tr>
<td>E</td>
<td>0.21</td>
<td>0.07</td>
<td>0.09</td>
<td>0.11</td>
<td>1.00</td>
<td>0.17</td>
<td>0.09</td>
<td>0.06</td>
<td>0.03</td>
<td>0.10</td>
<td>0.18</td>
</tr>
<tr>
<td>F</td>
<td>0.59</td>
<td>0.53</td>
<td>0.03</td>
<td>0.10</td>
<td>0.17</td>
<td>1.00</td>
<td>0.39</td>
<td>0.23</td>
<td>0.43</td>
<td>0.19</td>
<td>0.35</td>
</tr>
<tr>
<td>G</td>
<td>0.65</td>
<td>0.18</td>
<td>0.12</td>
<td>0.14</td>
<td>0.09</td>
<td>0.30</td>
<td>1.00</td>
<td>0.01</td>
<td>0.37</td>
<td>0.28</td>
<td>0.36</td>
</tr>
<tr>
<td>H</td>
<td>0.18</td>
<td>0.14</td>
<td>0.37</td>
<td>0.06</td>
<td>0.06</td>
<td>0.23</td>
<td>0.01</td>
<td>1.00</td>
<td>0.14</td>
<td>0.25</td>
<td>0.23</td>
</tr>
<tr>
<td>I</td>
<td>0.48</td>
<td>0.40</td>
<td>0.03</td>
<td>0.05</td>
<td>0.06</td>
<td>0.03</td>
<td>0.43</td>
<td>0.37</td>
<td>1.00</td>
<td>0.05</td>
<td>0.29</td>
</tr>
<tr>
<td>J</td>
<td>0.18</td>
<td>0.27</td>
<td>0.48</td>
<td>0.09</td>
<td>0.10</td>
<td>0.19</td>
<td>0.28</td>
<td>0.25</td>
<td>0.05</td>
<td>1.00</td>
<td>0.29</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.41</td>
<td>0.39</td>
<td>0.22</td>
<td>0.12</td>
<td>0.15</td>
<td>0.36</td>
<td>0.26</td>
<td>0.24</td>
<td>0.28</td>
<td>0.28</td>
<td>0.28</td>
</tr>
</tbody>
</table>

3.2 Experiment Design

We conduct a user study to evaluate the quality of the generated feedback-ladder. The annotators for the user study are recruited from several universities, with different knowledge levels of programming: students, CS instructors, and AI researchers. There are a total of 10 annotators. The user study consists of several different phases, detailed below.

**Eligibility determination phase:** The purpose is to determine that the annotators have sufficient programming skills required by the study. First, the annotators are shown a simple program in Java. Second, they are given a set of inputs for the given program. Third, they are asked to determine the output of the program on the given set of inputs. Failing this disqualifies them from participating in the study.

**Calibration phase:** The purpose is to calibrate the evaluation objective among annotators to align with our expectations. First, the annotators are shown examples of different feedback-ladder. Each example contains a problem statement, a submitted program, and a generated feedback-ladder. Second, they are asked various questions about the shown examples. The questions are related to identifying the proper level of a feedback text, evaluating qualitative measures, etc. If the annotator got any of the answers wrong, we show which questions are wrong but do not reveal the answer. They try again and choose different answers for the same questions. This phase ends when the annotators answer all the questions correctly.

**Evaluation phase:** The purpose is to evaluate the quality of the generated feedback-ladder for 15 total programs, i.e., 5 chosen problems with 3 different submissions each. First, the annotator is shown a problem description. Then they are shown a low-scoring student-submitted program and the generated feedback-ladder for it. Second, they are asked to rate two different metrics of each of the feedback on a 5-point Likert scale, which we detail below. This repeats two more times for the mid and high-scoring submissions. The whole thing repeats for four more problems.

**Evaluation Metrics:** We ask study annotators to rate the feedback of each level on the feedback-ladder on **Relevance** and **Effectiveness**. The relevance of feedback for a particular level is based on how well the feedback matches its level definition. 5 corresponds to a perfect match and 1 corresponds to no match. The effectiveness of a generated feedback for a particular level is how effectively it can help a student, according to the annotator’s judgment. 5 corresponds to highly effective feedback that should help the student fix their bug and 1 corresponds to feedback that does not help the student or may confuse them.

4. RESULTS AND DISCUSSION

4.1 Inter-Rater Agreement

Table 3 measures the inter-rater agreement between pairs of annotators in our study using Pearson correlation coefficient (PCC) [1], with average PCC values reported at the end of the row and column for the corresponding annotators. The overall average of all the pairwise PCC values is 0.28, which indicates moderate agreement across annotators.

4.2 Trends Found in the User Study

Figure 1 shows the trend for how annotators’ relevance and effectiveness ratings change across questions. There are four sets of bars, each for one question in our study. Each set contains the average score for each of the levels represented using a single bar, together with standard deviations. The scores are averaged over all scores from all the annotators and all three submissions for each question. Figure 2 shows the trend on the same evaluation metrics across different
feedback levels. Two key observations from these results: Quality of feedback for each level is consistent across questions. We see that the quality of feedback for lower levels is better than for higher levels in 4 out of 5 questions. In terms of relevance, our method can generate highly relevant lower-level feedback. However, for higher-level feedback, the relevance diminishes. Upon manual inspection, we observe that as the level increases, the feedback diverges from the level definition more and more. For example, level 4 feedback often has the whole program listed or contains a brand new program in the generated feedback-ladder, which violates the definition of level 4 feedback. On the other hand, the generated feedback is almost perfectly relevant at level 0 and level 1 since these levels require highly structured feedback. Overall, the method struggles to maintain relevance for levels 2, 3, and 4 a little bit more than levels 0 or 1. In terms of effectiveness, we see a similar trend as the one for relevance, except that the effectiveness at higher levels is much lower. In the annotators’ opinion, listing whole programs or replacing them with a new program has a more negative impact on effectiveness compared to relevance. An exception can be found between levels 1 and 2, where even though level 1 has low effectiveness, level 2 has higher effectiveness for questions 2, 3, and 4. This means that LLM-generated hints have higher quality than generated test cases.

Quality of feedback is higher for low-scoring submissions than high-scoring submissions. Among submissions with different scores (low-scoring: < 20%, mid-scoring: 40% - 60%, high-scoring: > 80%), we observe a clear trend that GPT-4 struggles to generate good feedback for higher-scoring submissions. This result is likely because finding a mistake in low-scoring submissions is much easier than doing so for higher-scoring submissions; high-scoring submissions are almost correct and often contain tiny mistakes that fail one or two corner test cases. Even human experts struggle to find these mistakes. We also find evidence of these situations upon manual inspection of the annotator ratings: The high-scoring submissions for Q1 and Q5 were so close to being correct that two of our annotators mistakenly believed they were correct. Therefore, they rated the generated feedback-ladder as irrelevant and ineffective at all levels.

5. CONCLUSION AND FUTURE WORKS

In our paper, GPT-4’s ability to generate feedback-ladders for programming assignments is examined. While promising, our user study revealed limitations in providing highly effective feedback, especially at advanced levels. However, the method remains valuable for teachers to tailor feedback to student proficiency levels, reducing workload by addressing mistakes in low-scoring submissions. Future research could include large-scale classroom studies for real-time interventions, personalized feedback models, making feedback personalized to each student’s knowledge [8] and dedicated LLM training for feedback generation to mitigate time and cost concerns associated with GPT-4.
6. REFERENCES


