

# Using GPT-4 to Provide Tiered, Formative Code Feedback

Ha Nguyen\*  
ha.nguyen@usu.edu  
Utah State University  
Logan, Utah, United States

Vicki Allan  
vicki.allan@usu.edu  
Utah State University  
Logan, Utah, United States

## ABSTRACT

Large language models (LLMs) have shown promise in generating sensible code explanation and feedback in programming exercises. In this experience report, we discuss the process of using one of these models (OpenAI's GPT-4) to generate individualized feedback for students' Java code and pseudocode. We instructed GPT-4 to generate feedback for 113 submissions to four programming problems in an Algorithms and Data Structures class. We prompted the model with example feedback (few-shot learning) and instruction to (1) give feedback on conceptual understanding, syntax, and time complexity, and (2) suggest follow-up actions based on students' code or provide guiding questions. Overall, GPT-4 provided accurate feedback and successfully built on students' ideas in most submissions. Human evaluators (computer science instructors and tutors) rated GPT-4's hints as useful in guiding students' next steps. Model performance varied with programming problems but not submission quality. We reflect on where the model performed well and fell short, and discuss the potential of integrating LLM-generated, individualized feedback into computer science instruction.

## CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Computing methodologies** → **Natural language generation**.

## KEYWORDS

computer science education, feedback, large language models

### ACM Reference Format:

Ha Nguyen and Vicki Allan. 2024. Using GPT-4 to Provide Tiered, Formative Code Feedback. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*, March 20–23, 2024, Portland, OR, USA. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3626252.3630960>

## 1 INTRODUCTION

Receiving effective feedback when programming can enhance students' learning, engagement, and motivation to pursue computer science [7, 16, 24]. However, instructors do not always have time to provide detailed feedback to every student in large courses. Additionally, instructors are sometimes too focused on the desired solutions that they fail to recognize meaningful attempts from students. Automated feedback can address these issues. Feedback comes in

the form of error suggestions [1], hints based on historical student submissions [25], and prompts to reflect on problem-solving steps [21]. Although promising, feedback systems can be expensive to develop as they often require large amounts of historical data for training [20]. Emergent research has explored the use of large language models (LLMs) to provide contextualized feedback without extensive input [3, 12, 30]. We explore the feasibility of using a LLM (OpenAI's GPT-4) to provide formative feedback and suggest follow-up actions that are logical next steps from students' code, rather than just providing one single, correct solution.

LLMs have learned language patterns from large amounts of data, and can now generate novel, human-like text [6]. These models have been applied to computer science (CS) contexts, to generate code, complete programming exercises, and automate code explanations [9, 12, 23, 32]. Assignments that ask students to submit code may no longer adequately assess learning, given LLMs' increasing capacities and availability. Students can use LLMs to generate answers, without learning the target concepts from completing the exercises themselves. Thus, instructors need to find ways to leverage the tools meaningfully. This is the motivation for our work.

Recent work has examined how LLMs can assess the accuracy of students' code [3, 18]. We extend this work and envision that LLMs can provide formative feedback for students. A common approach to giving feedback in large CS courses is to show sample solutions and point out common student errors. While this is easy, students do not get personalized feedback for their submissions. Our LLM feedback provides actions which are logical next steps from students' code. We term this approach *building on students' code*. The feedback takes different forms depending on the correctness of student submissions. If the submissions are close to correct, the feedback would suggest solutions that build on the logic and syntax of the original code. Meanwhile, if the submissions show fundamental misconceptions, the feedback would suggest starting over and include questions to guide students through the problem-solving steps. We explore the capacity of a LLM (GPT-4) in evaluating student submissions, based on the following questions:

**RQ1: To what extent can GPT-4 provide an accurate and useful evaluation of students' code?**

**RQ2: To what extent can the model build on students' code to give suggestions for improvement?**

We first describe the assignment context within an introductory course in algorithms and data structures (CS3). We outline the prompting strategies for GPT-4 and present a qualitative rubric to evaluate LLMs' feedback. Findings illustrate the promise of LLMs in assessing students' thinking and suggesting tailored next steps.

\*Authors contributed equally to this research.



This work is licensed under a Creative Commons Attribution-NonDerivs International 4.0 License.

## 2 RELATED WORK

### 2.1 Automated formative feedback

Automated programming feedback takes various forms. Summative feedback highlights syntactic errors and evaluates code performance through test cases [11]. Meanwhile, formative feedback systems remind students of task requirements, give examples to illustrate concepts, or provide hints and solutions to help students resolve errors and identify next steps [20, 21, 27, 28, 31].

Beyond feedback content, researchers have attended to *how* to frame feedback to positively influence learning and affective outcomes such as interest in CS [24, 26]. Effective feedback should be **tailored to students' level of understanding** [14]. Human tutors often iterate between different feedback forms when discussing programming solutions with students [26]. They review concepts, provide task-specific feedback about requirements, highlight gaps in students' solutions, and give advice about next steps drawing from students' code. However, few automated systems include all the feedback types observed in student-tutor interactions [20].

Effective feedback should also be **positive**—signaling to students when they have mastered a concept or performed the right steps [24, 29, 33]. Students perceive this type of feedback favorably [29]. They also master the targeted programming skills in less time when receiving positive feedback, compared to the condition without the feedback [13]. However, most automated feedback tools focus on providing feedback about students' mistakes (e.g., showing solution errors and test failures, [8, 15]). Acknowledging positive aspects of students' solutions and giving suggestions on how to proceed can increase their engagement with the feedback [14].

In sum, effective feedback should be positive and specific to students' code and level of understanding. Traditionally, automated systems to deliver such feedback require human input (pre-specified hints and solutions) or extensive training data (e.g., historical student solutions) [10, 24, 37]. This raises questions about the systems' adaptability to new assignment contexts. In the next section, we review the potential of LLMs in generating efficient feedback.

### 2.2 Potential of Large Language Models

Large Language Models (LLMs) like GPT-4 are neural networks that have been trained on large text corpora, and can now replicate language patterns [6]. In CS education contexts, these models have been used to generate code based on natural language prompts [9, 19, 30], create code explanations [23], support debugging processes [2], and provide feedback on code [3, 18]. For instance, researchers used GPT-3 (an earlier version of GPT-4) to provide evaluations of students' code from seven programming questions [3]. GPT-3 performed reasonably well: it accurately evaluated code correctness in 71.13% of the submissions, and correctly suggested improvements in 61.95% of submissions. However, the model showed high variability between questions [3]. Meanwhile, [18] found that the performance of more recent LLMs (GPT-3.5, Codex) varied with the types of errors in student code. For example, the models were more successful in identifying errors related to iteration than conditionals [18]. While the models successfully identified issues in 82% of submissions, 48% of the feedback reported non-existing errors. Both types of errors can negatively impact learning. Students might not learn the target concepts, when the

feedback fails to identify issues in their approaches. Meanwhile, inefficient or erroneous suggestions might increase learners' cognitive load and lead them down less productive debugging paths [22, 30, 35]. Thus, it is important to explore whether more recent LLMs might provide more accurate feedback.

We used a tiered approach for the formative feedback, beyond evaluating code accuracy. We prompted the model to select from three actions: (1) build on students' code to provide next steps toward solutions, (2) provide instructor-generated, example solutions, or (3) start over with guiding questions about the concepts. Deciding on the most appropriate actions is difficult even for human instructors. It requires understanding of the underlying intent of students' code and pseudocode (which are sometimes ambiguous) and knowledge of the various solutions to a programming problem. Our work takes steps to explore GPT-4's capacity in this context.

## 3 CONTEXT AND EXAMPLE GPT-4 FEEDBACK

This work takes place in an introductory course about algorithms and data structures (CS3) that the second author teaches. The class has an enrollment of 80 students per semester. Students complete weekly, 15-minute online quizzes to practice course concepts. The quiz format encourages design, prevents students from seeking outside help, and limits the time investment.

Students are traditionally evaluated in two ways. First, points are awarded for completion. Second, the instructor randomly selects a third of the submissions to grade and provides more detailed feedback for each quiz. This ensures that each student will be graded at least once. Following each quiz, the instructor releases a lecture video to discuss common mistakes and possible solutions.

We envision that LLMs can provide more frequent, formative feedback on students' submissions—something that instructors are not able to provide for large courses. In this paper, we test the model's capacity with four programming questions focusing on recursion. Recursion is traditionally challenging for novice programmers, as they need to consider different local values when examining the same function. Even though students have encountered recursion in previous courses (CS 1 and 2), many have not mastered the concept when they get to CS3. Gaining practice and receiving formative feedback on their recursive thinking is invaluable. The quizzes are given early in the semester as building blocks for later programming assignments involving balanced search trees. They allow students to focus on concepts rather than syntax in a low-stakes setting. In addition to recursive thinking, students need to master base cases, using return values, and recursive data structure usage. The quiz questions are as follows:

**Problem 1:** Write the recursive code to efficiently find the number of 1s in a sorted binary array. The aim is to beat  $O(n)$  complexity.

**Problem 2:** You are given a binary tree. Write the Java code to determine if it is a leftist tree. A leftist tree is one in which a node in the tree never has a right child without a left child.

**Problem 3:** A teacher gives a 100 point exam to  $N$  students. The teacher wants to determine the median score. The code needs to have the best complexity possible. (a). Write the code to determine the median. (b). What is the complexity of your algorithm?

**Student's Code** (student grade B):

```
public numOnes(int[] binary) {
    if binary.length() == 1 {return binary[0]}
    else if binary.length() == 0 {return 0}
    marker = binary.length()/2
    // if .length() function is O(n) this is all worthless I guess

    if binary[marker] == 1 {
        return numOnes(binary[:marker])+marker // probably
        not the right syntax since I've been working on Python
    } else {
        return numOnes(binary[marker:])
    }
}
```

**GPT-4's suggested code:**

```
public int numOnes(int[] binary) {
    return numOnes(binary, 0, binary.length - 1);
}

private int numOnes(int[] binary, int left, int right) {
    if (left > right) return 0;
    int marker = left + (right - left) / 2;
    if (binary[marker] == 1) {
        return numOnes(binary, left, marker - 1) + right - marker + 1;
    } else {
        return numOnes(binary, marker + 1, right);
    }
}
```

**GPT-4's Feedback**

**"Evaluation of concept":** "Your approach of implementing a binary search to find the first occurrence of 1 is the correct direction. You correctly identified that dividing the array into two halves and determining which half to proceed with based on the midpoint value is a good way to handle this problem. However, the details in your implementation are not quite correct. In the case when the mid value is 0, you should search in the right half, but it should start from mid + 1, not mid.",

**"Evaluation of syntax":** "There are quite a few syntax errors in your code. In Java, we use parentheses '()' for if statements, method calls, and array access. Also, the way to get the length of an array in Java is 'array.length', not 'array.length()'. Additionally, the array slicing syntax you used is from Python and does not apply in Java. Lastly, the data types for 'marker' and the function 'numOnes' are missing.",

**"Evaluation of time complexity":** "If the logic was implemented correctly, your solution would have a complexity of  $O(\log n)$  because you're using a binary search approach. This is the correct approach considering the sorted binary array.",

**"Your strategy** to build on student's code, give the correct code, or start over": "Given the current solution's conceptual and syntax issues, it might be best to **start over**. I will provide a corrected code based on the initial approach."

**"Hint":** "Given that the array is sorted, you can take advantage of binary search. Try finding the first occurrence of 1 in the array. Once you find the first 1, you can calculate the number of 1's by subtracting its index from the length of the array."

**Figure 1: Example GPT-4's feedback for a B-graded submission. GPT-4 built on the student's approach.**

**Problem 4:** Given a set of  $N$  non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to a given sum,  $K$ . Each item can only be used once. This problem is known to be intractable. Give two solutions: one  $O(2^N)$  and the other  $O(NK)$  in complexity.

LLMs like GPT-4 can take text instruction as prompts to produce text output, based on predictions of what might likely follow the prompts. The example quiz questions above can be used as part of the prompt. Additionally, our prompt for each quiz question included two examples of possible correct solutions, two samples of student code, and two examples of feedback for the provided code. This approach is called few-shot learning, allowing the pre-trained GPT models to generalize over new text patterns [6]. Our prompt also included instructions for the tiered feedback:

**Student's Code** (student grade C):

```
int total
public static int findOnes(int[] binaryArr, int n){
    if(binaryArr[n] == 1){
        total++;
    }

    if(n == binaryArr.length) return;
    findOnes(binaryArr, n+1);
}
```

**GPT-4's suggested code:**

```
int total = 0;

public int findOnes(int[] binaryArr, int n){
    if(n == binaryArr.length) return total;
    if(binaryArr[n] == 1){
        total += binaryArr.length - n;
        return total;
    }
    return findOnes(binaryArr, n+1);
}
```

**GPT-4's Feedback**

**"Evaluation of concept":** "Your concept using recursion to count the number of 1's is technically correct, but it is not efficient for a sorted binary array. The concept doesn't consider that once we encounter the first 1, the rest of the array will be filled with 1's due to the array being sorted. The current concept results in scanning through the entire array which is not needed.",

**"Evaluation of syntax":** "There are multiple syntax errors. The variable 'total' is declared but its data type is missing. The recursive function 'findOnes' does not return a value. Also, the function needs to return the value of 'total' for the result.",

**"Evaluation of time complexity":** "The code, in its current form, would have a time complexity of  $O(n)$  as it scans the entire array to count the 1's. This is not the most efficient solution for the given problem. With a sorted binary array, we can achieve a time complexity of  $O(\log n)$  using a binary search approach.",

**"Your strategy** to build on student's code, give the correct code, or start over": "Given the current solution's conceptual and syntax issues, it might be best to **start over**. I will provide a corrected code based on the initial approach."

**"Hint":** "Given that the array is sorted, you can take advantage of binary search. Try finding the first occurrence of 1 in the array. Once you find the first 1, you can calculate the number of 1's by subtracting its index from the length of the array."

**Figure 2: Example GPT-4's feedback for a C-graded submission. GPT-4 fixed the code and hinted at the right solution.**

You are a coaching tutor. Provide evaluation for the student's code. Follow the following logic: 1. If student's code is correct but not efficient, provide evaluation of code and show correct, efficient solution. 2. If student's code is incorrect and inefficient, correct their solution with code based on student's code but don't show the desired solution. 3. If student's code is not usable, explain that in your reasoning and start over with guiding questions for students. You will be given the student's code as input. "Student's Code": "...", Follow this format for your output: "Evaluation of concept": "...", "Evaluation of syntax": "...", "Evaluation of time complexity": "...", "Your strategy to build on student's code, give the correct code, or start over", "Correct Code": "...", "Hint": "...",

Figures 1 and 2 show example student's code and snapshots of GPT-4's feedback. Note how the model was able to point out what the students did correctly (i.e., positive feedback). It built on the student's approach toward an efficient solution (Figure 1) or hinted at the correct solution (Figure 2).

## 4 METHODS

### 4.1 Prompt development

We made API requests to OpenAI’s GPT-4 model (temperature = 1; maximum tokens = 1000, frequency penalty = 0). Each request included the prompt for tiered feedback (outlined in section 3) and one student submission. In total, the model provided feedback on 113 submissions to four practice problems in past years (28 submissions/ problem on average). The research was approved by the university’s Institutional Review Board (#13747). Submissions often contained multiple issues, including logic errors (e.g., missing base cases, missing conditionals), complexity errors (e.g., incorrectly assessing code efficiency), syntax errors (e.g., missing brackets, mixing Java and Python syntax), and incomplete solutions.

The prompt design went through several iterations. We started by having the model evaluate code accuracy. We then developed more fine-grained instructions to focus on the correctness of conceptual understanding, syntax, and time complexity evaluation. Given that responses showed different levels of accuracy, we further evaluated the model’s capacities to generate tiered feedback. Specifically, we prompted the model to evaluate the quality of responses and decide on one of three actions: (1) provide efficient solutions while recognizing student’s ideas (when student’s code is correct but inefficient), (2) build on students’ code (when their code is incorrect but fixable), or (3) start over. We asked the model to provide the rationale for its decisions.

We used the prompt with different GPT-based models (GPT-3.5, GPT-4 with no example of feedback, and GPT-4 with examples). For each model, we generated feedback for 15 submissions to problems 1-3 (5 submissions per problem). The second author (a CS instructor) manually evaluated the feedback. Of those, GPT-4 with examples showed the highest consistency. This model provided correct evaluation actions (e.g., building on student’s submissions, starting over) 87% of the time. In comparison, GPT-4 with no examples provided correct actions 67% of the time and GPT-3.5 60% of the time. We used GPT-4 with examples (i.e., few-shot learning) to evaluate the remaining submissions.

### 4.2 Analytic strategies

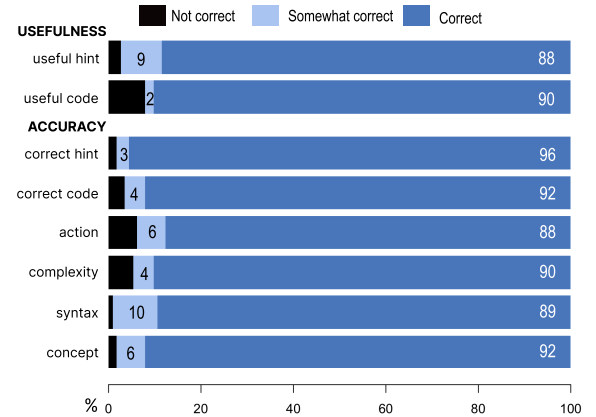
We developed a rubric to evaluate the feedback generated by GPT-4 (Table 1). The rubric focused on feedback accuracy and usefulness for code improvement. Two evaluators (a CS instructor and a senior CS undergraduate) used the rubric to rate a subset of 20 feedback separately and achieved substantial agreement (percent agreement 75%-100%; Gwet’s AC1 range .72-1, [17]).<sup>1</sup> The evaluators each rated half of the remaining feedback. Evaluators wrote short notes in one to two sentences describing where GPT-4 seemed to perform well, make mistakes, or give too much information.

To address **RQ1—accuracy and usefulness of GPT-4’s evaluations**, we reported the evaluators’ ratings by problems and student grades. We ran Chi-square tests of independence, to examine whether there existed differences in performance across problems and submission quality. Additionally, through thematic analysis [5], we organized evaluators’ notes into themes to illustrate the model’s utility and where it might fall short.

<sup>1</sup>We reported these statistics, as Cohen’s kappa is sensitive to imbalanced distributions. In our case, evaluators rated GPT-4’s feedback as correct >90% of the time.

**Table 1: Rubric for GPT-4 Evaluations**

Code	Description
<b>Accuracy</b>	
Concept	Whether evaluation of students’ approaches, task understanding, and logic is correct
Syntax	Whether syntax evaluation is correct
Time Complexity	Whether complexity evaluation is correct
Action	Action to build on code or start over is correct
Code	Whether suggested code is correct
Hint	Whether hint is conceptually correct
<b>Usefulness</b>	
Code	Suggested code builds on student’s code or shows more efficient solution
Hint	Gives useful information about next steps
<b>Notes</b>	
Open-ended evaluation notes	
Ratings: correct/useful, somewhat correct/useful, incorrect/not useful	



**Figure 3: Ratings of GPT-4’s feedback**

For **RQ2—the extent to which the model built on students’ code**, we reported the percentage of GPT-4’s code that successfully built on students’ submissions. We focused on cases where students’ code was usable and GPT-4’s suggestions showed improvement, compared to cases where GPT-4’s suggestions were not optimal.

## 5 RESULTS

### 5.1 RQ1: Accuracy and usefulness of feedback

**5.1.1 Rubric ratings.** We generated feedback on 113 students’ submissions in four programming problems in data structures. We evaluated the feedback accuracy regarding concepts, syntax, and time complexity with three ratings: incorrect, somewhat correct, and correct.

**GPT-4 generally provided accurate feedback** (Figure 3). The model provided correct evaluations of students’ conceptual understanding in 92.04% of submissions, syntax in 89.38% of submissions,

and time complexity in 90.27% of submissions. These results were promising, given that submissions contained a mix of code, pseudocode, and text explanations (that were not always clearly separated from code). We further evaluated the accuracy of the model's suggested code. The model performed well in this aspect, providing correct code for 92.03% of submissions and somewhat correct code (with some syntax errors) for 4.42% of submissions. Finally, the provided hints for next steps were conceptually correct in 95.58% of submissions (somewhat correct: 2.65%).

**The feedback was useful in guiding students' next steps**, as determined by human evaluators who were CS instructors and tutors. Overall, the model provided useful code (90.27% of submissions were useful; 1.77% somewhat useful, 7.96% not useful). Similarly, the follow-up hints were overall useful in 88.50% of submissions (8.85% somewhat useful, 2.65% not useful).

**We saw variation in the model's performance across programming problems.** Chi-square test of independence suggested different distributions across problems for hint's usefulness,  $\chi^2(6, N = 113) = 15.22, p = .02$ . Post-hoc tests with Bonferroni adjustments revealed significant differences between programming problem 1 and the others, with feedback in problem 1 receiving more "somewhat helpful" ratings for the generated hints. An explanation is that the hints for this problem were sometimes repetitive. Additionally, GPT-4 did not always suggest the most efficient solutions. We note that even human graders were occasionally conflicted in deciding whether to build on a correct but inefficient solution, or abandon the student solution to promote the ideal solution.

Finally, we evaluated GPT-4's feedback in relation to the quality of students' submissions (letter grades A, B, C and below). We **did not find significant variation with submission quality**.

**5.1.2 Qualitative observations: When did the Model Perform Well?** Evaluators' notes revealed that **the feedback sometimes uncovered ideas that instructors might miss**. For example, in feedback for problem 3 (finding the median of test scores), GPT-4 successfully recognized when students were going for less efficient, but correct solutions, e.g.,  $O(n^2)$  by employing a sort instead of  $O(n)$  using an array of frequencies. The instructors' feedback sometimes missed these solutions, particularly if the sort was poorly implemented.

**GPT-4's feedback also conveyed positive tone**, for example:

While your thoughts on implementing a heap for this task show that you're thinking critically about the problem, it's actually not necessary to use such a complex data structure. The task can be solved using simpler methods like array manipulation and scanning.

This feedback acknowledged the student's thought process, while guiding them toward the right solutions. GPT-4 followed up with a series of guiding questions: "Let's start by understanding the problem. How would you read the scores from a file? Can you think of a way to store the frequency of each score? How would you determine which score is the median?" In comparison, instructors are less likely to provide this type of feedback frequently. In lecture videos discussing the solutions, they may focus on efficient solutions and highlight submissions with serious deficiencies. The model showed a more forgiving, positive approach, which can build students' confidence and understanding [24, 29].

Finally, **student-GPT-4's combined efforts resulted in additional solutions, compared to instructor-created solutions**. In problem 1 (finding the number of 1s in a sorted binary array), the sample solutions focused on checking if the middle element was a 1. Meanwhile, some students sequentially counted the 1's and looked to see if the first element was a 1. GPT-4 successfully built on students' code and created a different base case. While the model solution recursively called the count method on the first half or the second half of the array, GPT-4 allowed the users to call the recursive solution on both halves of the array, but short-circuit the evaluation if the first item in the array was a 1.

**5.1.3 Qualitative Observations: When did the Model Fall Short?** We documented cases where GPT-4 could have given better feedback. Findings echoed research with earlier LLMs [3, 18]: GPT-4 stated non-existing errors, although to a lesser extent than prior models.

- Concept misvaluation: incorrectly stating that approaches were correct when they were not (1.77% of submissions).
- Syntax misvaluation: stating non-existing errors (4.42%).
- Misfocus: failing to highlight key problems, and instead focusing on minor syntax or efficiency issues (6.19%).
- Correct code recognition: Given that 12.38% of students' code was already correct, GPT-4 recognized that the code did not need to be fixed in 78% of those cases but suggested unnecessary changes in 22% of the cases.
- Local correction: only providing feedback on part of the answer, instead of the entire submission (4.42%).
- Too much information: providing entire solutions instead of guiding questions when students showed fundamental misconceptions; hints were too helpful (4.42%).

## 5.2 RQ2: Building on students' submissions

The model suggested appropriate actions to start over or build on ideas in 87.61% of submissions. We found the following cases:

- Submissions had fundamental errors in approaches, and GPT-4 correctly suggested starting over (27.43% of submissions).
- Submissions could be built on, and GPT-4 suggested incremental improvements and more efficient solutions (17.70%).
- Submissions were good, and GPT-4 made minor changes like fixing syntax errors or leaving code unchanged (35.40%).
- GPT-4 correctly chose to build on submissions, but the suggested code was not optimal (7.08%).

And cases where human's and GPT-4's actions differed:

- Submissions had fundamental errors, but the model chose to build on students' code (2.65%).
- Submissions could be built on, but GPT-4 started over (9.73%).

Overall, GPT-4 was able to provide meaningful feedback building on students' ideas (incrementally or with minor edits). It correctly suggested that students start over most of the time, when their code contained fundamental errors. Tailoring feedback to learners' submissions may lead to positive impacts on students' motivation and conceptual understanding [24, 29].

## 6 DISCUSSION

We explored LLMs' ability to provide formative feedback on submissions that included code, pseudocode, and text explanations.



We provided concrete examples of prompts for LLMs to generate formative feedback. Findings suggest that GPT-4 with few-shot learning (instructed with only two sets of example student code and feedback) generally performed well. The model correctly evaluated conceptual understanding, syntax, and time complexity in most submissions. Additionally, the suggested code was generally correct, and the hints were perceived as useful by domain experts in guiding students' thinking. These results are promising, given the significant time it takes to provide tailored feedback to students. For example, it takes a few seconds to generate GPT-4's feedback for a submission, compared to 3–5 minutes for a human grader to achieve the same level of feedback. Further, we note GPT-4's ability to recognize students' approaches and build on ideas in positive tone. Below, we reflect on ways to integrate LLMs into introductory CS courses and outline future directions for research and practice.

### 6.1 Feedback for iterative code improvement

We recognize that the generated feedback was not perfect. We note instances where the feedback showed inconsistencies or mistakes, such as incorrectly interpreting code or missing key errors in students' understanding. However, in evaluating GPT-4's responses, we realized that human feedback was also not correct or consistent 100% of the time. In fact, our qualitative findings illustrated how the model picked up ideas that the human evaluators had missed.

We propose two directions to use LLMs-generated feedback in instruction and **invite students to critique the models' feedback**. First, students can use the feedback to improve their solutions and submit runnable code in a follow-up assignment. This process might provide students with additional opportunities to practice course concepts. Students might be encouraged to document their thought processes in making sense of and iterating on the feedback. Iterative learning—reflecting on feedback and incrementally improving code—may enhance learning and attention to code quality [4, 36]. Second, students can receive feedback from LLMs in real time. Emergent research has explored how novice programmers use LLMs to write live code [19, 30]. Analyses of student-LLM feedback interaction can attend to (1) how students learn from and iterate on the feedback, (2) how much time and effort students spend on comprehending the feedback, and (3) the best feedback types for different concepts and levels of understanding.

Although we did not find significant variation in performance across submission quality, we note considerations about **the equitable integration of LLMs to provide feedback**. Students with emergent understanding might be more likely to get confused when the feedback is inaccurate [18, 30]. Researchers have also noted that LLMs generally have a confident, assertive tone (even when they are wrong [3]). Thus, instructors might consider framing the model's feedback as suggestions rather than evaluations. Specifically, gamification approaches might reduce frustration when receiving incorrect feedback [22]. LLMs-generated feedback can be prefaced with instructions such as: "The AI-generated feedback is correct around 90% of the time. Does your feedback lie in the 10% error segment? Earn bonus points by correctly identifying the erroneous suggestions". Additionally, instructors may consider human-in-the-loop safeguards, such as combining LLMs' feedback with peer reviews, to encourage critical evaluations of the feedback.

### 6.2 Designing feedback prompts

Our work illustrates the potential of GPT-4 with few-shot learning. Different from prior work with GPT-3 [3], we did not find variability in feedback accuracy across programming questions. We also found fewer cases of the model identifying non-existing errors, compared to research with GPT-3.5 [18]. While we recognize that LLMs are getting updated rapidly (and future models may outperform GPT-4), our work illuminates the **affordances of tuning LLMs with well-designed instructions and desired output**.

Designing better feedback prompts is a future direction. We found instances where the model's suggested actions were not optimal. For next steps, we will experiment with explicit prompts and assignment-specific examples that define when a student's answer is efficient, inefficient but fixable, or unusable. Researchers have discussed the possibility of adapting feedback to students' preferences and knowledge levels [23], or adding messages with explicit positive affects [24]. Future research can investigate these directions, drawing from educational research on feedback [14, 33] and prompt engineering strategies [18, 34]. For example, scholars have prompted LLMs to take on different roles, as feedback providers and students reasoning with the provided feedback [34]. The feedback LLM then iterated on its strategies, given input from the student LLM. These investigations can include questions such as how to frame and follow up on feedback [18].

### 6.3 Limitations

A limitation of this research is that we focused the analysis on programming problems about data structures in one programming language (Java). Future work can validate LLMs' performance across a broader range of programming concepts, difficulty, and programming languages. Additionally, LLMs like GPT-4 are probabilistic, meaning that the same model can generate slightly different responses for the same input. Finally, our evaluations of the model's usefulness were based on feedback from CS instructors and tutors and not students. Our next step is to analyze students' learning outcomes from interacting with LLMs-generated versus human-generated feedback. We will also evaluate students' willingness to share their code for real-time LLM-generated feedback and their perceptions of the model's utility in supporting debugging, concept understanding, and engagement in the discipline.

## 7 CONCLUSION

With this work, we provide examples of how to prompt LLMs like GPT-4 to generate formative feedback based on students' current levels of understanding. In most cases, the feedback correctly considers multiple code aspects (conceptual understanding, syntax, time complexity) and builds on students' ideas. We highlight areas where the model might be useful in large instructional settings, such as framing feedback positively and recognizing students' attempts. We highlight variations in GPT-4's performance across programming problems and discuss ways to efficiently integrate these tools into instruction.

## ACKNOWLEDGMENTS

We thank Peter Fowles for helping with evaluating the model.

## REFERENCES

- [1] Umair Z Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. 139–150.
- [2] Erfan Al-Hossami, Razvan Bunescu, Ryan Teehan, Laurel Powell, Khyati Mahajan, and Mohsen Dorodchi. 2023. Socratic Questioning of Novice Debuggers: A Benchmark Dataset and Preliminary Evaluations. 709–726.
- [3] Rishabh Balse, Bharath Valaboju, Shreya Singhal, Jayakrishnan Madathil Warriem, and Prajish Prasad. 2023. Investigating the Potential of GPT-3 in Providing Feedback for Programming Assessments. 292–298.
- [4] Martin Barker and Michelle Pinard. 2014. Closing the feedback loop? Iterative feedback between tutor and student in coursework assessments. *Assessment & Evaluation in Higher Education* 39, 8 (2014), 899–915. Publisher: Taylor & Francis.
- [5] Virginia Braun and Victoria Clarke. 2012. *Thematic analysis*. American Psychological Association.
- [6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, and Amanda Askell. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [7] Albert T Corbett and John R Anderson. 2001. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. 245–252.
- [8] Draylson Micael De Souza, Jose Carlos Maldonado, and Ellen Francine Barbosa. 2011. ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. *IEEE*, 1–10.
- [9] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. 1136–1142.
- [10] Ta Nguyen Binh Duong, Lwin Khin Shar, and Venky Shankaraman. 2022. AP-Coach: formative feedback generation for learning introductory programming concepts. *IEEE*, 323–330.
- [11] Stephen H Edwards and Krishnan Panamalai Murali. 2017. CodeWorkout: short programming exercises with built-in data collection. 188–193.
- [12] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A Becker. 2023. My AI Wants to Know if This Will Be on the Exam: Testing OpenAI’s Codex on CS2 Programming Exercises. 97–104.
- [13] Davide Fossati, Barbara Di Eugenio, STELLAN Ohlsson, Christopher Brown, and Lin Chen. 2015. Data driven automatic feedback generation in the iList intelligent tutoring system. *Technology, Instruction, Cognition and Learning* 10, 1 (2015), 5–26. Publisher: Old City Publishing. 628 North 2nd Street, Philadelphia, PA 19123.
- [14] Graham Gibbs and Claire Simpson. 2005. Conditions under which assessment supports students’ learning. *Learning and teaching in higher education* 1 (2005), 3–31. Publisher: University of Gloucestershire.
- [15] Sebastian Gross, Bassam Mokbel, Barbara Hammer, and Niels Pinkwart. 2015. Learning Feedback in Intelligent Tutoring Systems: Report of the FIT Project, Conducted from December 2011 to March 2015. *KI-Künstliche Intelligenz* 29 (2015), 413–418. Publisher: Springer.
- [16] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-driven feedback: Results from an experimental study. 160–168.
- [17] Kilem Li Gwet. 2008. Computing inter-rater reliability and its variance in the presence of high agreement. *Brit. J. Math. Statist. Psych.* 61, 1 (2008), 29–48. Publisher: Wiley Online Library.
- [18] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutchme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the Responses of Large Language Models to Beginner Programmers’ Help Requests. *arXiv preprint arXiv:2306.05715* (2023).
- [19] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. 1–23.
- [20] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018), 1–43. Publisher: ACM New York, NY, USA.
- [21] Amy J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. 151–158.
- [22] Ben Lafreniere, Tanya R. Jonker, Stephanie Santosa, Mark Parent, Michael Glueck, Tovi Grossman, Hrvoje Benko, and Daniel Wigdor. 2021. False Positives vs. False Negatives: The effects of recovery time and cognitive costs on input error preference. 54–68.
- [23] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. 2022. Generating diverse code explanations using the gpt-3 large language model. 37–39.
- [24] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W Price, and Tiffany Barnes. 2020. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. 194–203.
- [25] Jessica McBroom, Irena Koprinska, and Kalina Yacef. 2021. A survey of automated programming hint generation: The hints framework. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–27. Publisher: ACM New York, NY.
- [26] Claudia Ott, Anthony Robins, and Kerry Shephard. 2016. Translating principles of effective feedback for students into the CS1 context. *ACM Transactions on Computing Education (TOCE)* 16, 1 (2016), 1–27. Publisher: ACM New York, NY, USA.
- [27] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)* 22, 3 (2022), 1–40. Publisher: ACM New York, NY.
- [28] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. 195–204.
- [29] Ann Poulos and Mary Jane Mahony. 2008. Effectiveness of feedback: The students’ perspective. *Assessment & Evaluation in Higher Education* 33, 2 (2008), 143–154. Publisher: Taylor & Francis.
- [30] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It’s Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *arXiv preprint arXiv:2304.02491* (2023).
- [31] Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: towards intelligent tutoring in novice programming environments. 483–488.
- [32] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. 27–43.
- [33] Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189. Publisher: Sage Publications.
- [34] Shashank Sonkar, Lucy Liu, Debshila Basu Mallick, and Richard G Baraniuk. 2023. CLASS Meet SPOCK: An Education Tutoring Chatbot based on Learning Science Principles. *arXiv preprint arXiv:2305.13272* (2023).
- [35] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. 1–7.
- [36] Yi-Xiang Yan, Jung-Pin Wu, Bao-An Nguyen, and Hsi-Min Chen. 2020. The impact of iterative assessment system on programming learning behavior. 89–94.
- [37] Rui Zhi, Samiha Marwan, Yihuan Dong, Nicholas Lytle, Thomas W Price, and Tiffany Barnes. 2019. Toward Data-Driven Example Feedback for Novice Programming. *International Educational Data Mining Society* (2019). Publisher: ERIC.