Next-Step Hint Generation for Introductory Programming Using Large Language Models

Lianne Roest
Utrecht University
Utrecht, The Netherlands
lianneroest@gmail.com

Hieke Keuning
Utrecht University
Utrecht, The Netherlands
h.w.keuning@uu.nl

Johan Jeuring
Utrecht University
Utrecht, The Netherlands
j.t.jeuring@uu.nl

ABSTRACT

Large Language Models possess skills such as answering questions, writing essays or solving programming exercises. Since these models are easily accessible, researchers have investigated their capabilities and risks for programming education. This work explores how LLMs can contribute to programming education by supporting students with automated next-step hints. We investigate prompt practices that lead to effective next-step hints and use these insights to build our StAP-tutor. We evaluate this tutor by conducting an experiment with students, and performing expert assessments. Our findings show that most LLM-generated feedback messages describe one specific next step and are personalised to the student's code and approach. However, the hints may contain misleading information and lack sufficient detail when students approach the end of the assignment. This work demonstrates the potential for LLM-generated feedback, but further research is required to explore its practical implementation.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**.

KEYWORDS

Next-step hints, automated feedback, learning programming, Large Language Models, Generative AI

ACM Reference Format:
Lianne Roest, Hieke Keuning, and Johan Jeuring. 2024. Next-Step Hint Generation for Introductory Programming Using Large Language Models. In *Australian Computing Education Conference (ACE 2024), January 29–February 02, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3636243.3636259

1 INTRODUCTION

Over the last decades, researchers have been developing a variety of digital tools and systems that support students in learning how to program. Most of these tools provide automated feedback on student solutions to exercises [19]. This feedback may include hints for code edits, references to relevant concepts, or suggestions for reading material. In this paper we will focus on *next-step hints*. These hints help students with how to proceed when they are stuck while solving a programming exercise.

Next-step hints can be generated automatically by comparing possibly incomplete student programs with model solutions [18], or with historical data from other students [36, 38]. Some issues with data-driven approaches are that they often require a large amount of data. Furthermore, these hints are often exact directions for what a student should do, revealing an answer and thus interfering with learning opportunities [37]. Adding explanations can improve students' understanding of the hints and perceived relevance [27]. However, handcrafting explanations is a lot of work, nullifying the advantage of data-driven approaches regarding development time.

In this paper we examine how large language models (LLMs) can be used to generate next-step hints. When using LLMs, we don't need historical student data or model solutions anymore. The performance of LLMs has increased significantly in the last couple of years. LLMs are trained on enormous amounts of data and have shown to be very good at specific tasks, such as generating text, images, and code [4]. Especially with the arrival of the models from OpenAI (GPT-3/4, Codex), LLMs have received much attention, both in and outside the academic world. Studies show that they are very good at solving introductory programming assignments [10, 33]. As these models are now also available for students, there are concerns regarding plagiarism, integrity, and learning [2, 10].

Novice programmers may be particularly susceptible to the 'dangers' of using LLMs as their pair programmers. They do not have the skills yet to understand code and recognise code of good quality, and might not know how to design effective prompts [9]. Moreover, students might overestimate their coding abilities using LLMs, leading to over-reliance and reduced learning [33].

In this study, we focus on introductory Python exercises, as Python is often used in novice programming courses. We investigate how to design prompts for LLMs to produce next-step hints, and enhance them with additional information (e.g. explanations). We want to go beyond just edit instructions, as often generated with data-driven methods. Based on our findings, we built the StAP-tutor (**St**ep **A**ssisted **P**rogramming tutor). We evaluate the hint quality with an experiment with students and an expert assessment. We investigate the following research question and two subquestions:

- **RQ1** To what extent can we use LLMs to generate informative and effective next-step hints for Python introductory programming exercises?
- **SQ1** What prompt characteristics are suitable for generating effective next-step hints with LLMs?
- **SQ2** What are students' and experts' perceptions of the quality of LLM-generated next-step hints?

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

ACM ISBN 979-8-4007-1619-5/24/01…$15.00
https://doi.org/10.1145/3636243.3636259

Section 2 discusses related work on effective feedback, generating automated feedback, and using LLMs in programming education. We present a short overview of the method in Section 3, with a more in-depth discussion on prompt-engineering in Section 4. Section 5 describes the tool and its evaluation. Section 6 discusses limitations and directions for future research. Section 7 concludes.

2 RELATED WORK

2.1 Automated feedback

Digital learning tools have been developed across multiple domains. Many support teaching topics within computer science and programming [28]. Providing (automated) feedback [6] is an important aspect of these tools.

2.1.1 Providing effective feedback. Feedback is essential for learning [13]. It can address mistakes or misconceptions, improve motivation, or provide guidance when a student is stuck. *Formative* feedback consists of information or learning activities that support student learning [15]. *Summative* feedback consists of assessment activities, resulting in grades that evaluate a student's performance.

Different variables impact the effectiveness of formative feedback [40]. *Elaborated* feedback helps students understand why something is wrong and can guide students in the right direction with explanations or tips. However, if feedback is too long, learners pay no attention to it, rendering it useless. Another essential factor is *timing*. Feedback can be provided by intervening or waiting until a student requests feedback themselves or finishes a task. The *nature* of feedback can be positive or negative. Positive feedback reinforces what students are doing well, for example, to emphasise a student is correctly implementing task requirements. Both positive and negative feedback can have beneficial effects on learning [13].

Defining effective feedback is not easy, and students and teachers have sometimes different ideas about it [5]. While teachers believe that timing, modalities and connected tasks are important, students prefer detailed feedback personalised to the student's work.

2.1.2 Feedback on programming tasks. Narciss [29] identified five main categories for the content of feedback: Knowledge About Task Constraints, Knowledge About Concepts, Knowledge About Mistakes, Knowledge About How to Proceed, and Knowledge About Meta-cognition. For the domain of programming, Keuning et al. [19] conducted a systematic review of automated programming feedback in which they extended Narciss' classification for this domain. They found that most programming learning tools provide feedback with knowledge about mistakes by implementing automated testing. However, Hao et al. [12] found that feedback of this type is not as effective as more informative and detailed feedback. Only presenting mistakes requires students to interpret the results themselves, which may interfere with learning from their misconceptions.

This paper focuses on generating feedback on how to proceed. These next-step hints can take different forms, such as suggestions, questions or instructions. A hint's goal can be to correct an error or guide a student to take a next step towards a correct solution.

Several techniques generate automated programming feedback. Keuning et al. identified three universal techniques (model tracing, constraint-based modelling, and data analysis) and five programming-specific (e.g. dynamic code analysis, and static analysis).

Classical approaches such as model tracing have been used for next-step hints, but they require a significant amount of manual labour [1, 18]. Every knowledge component, production rule, or constraint has to be defined by experts. Especially for programming, where tasks often have different semantic or syntactic solutions, this can be very time-consuming. Data analysis overcomes this issue by employing algorithms that use large existing datasets to learn patterns or strategies that lead to correct solutions.

2.1.3 Data-driven next-step hints. Data-driven approaches for programming often focus on next-step hints [36]. A data-driven approach usually operates as follows: 1) compare the current student state (from the moment of the hint request) to a desired target state, often a correct solution; 2) identify differences or required edits to transform the current state into a target state; 3) extract one or more next steps [26, 38].

As data-driven methods avoid the need for many model solutions, they are considered less time-consuming than traditional techniques. Another advantage is they can generate hints for never-before-seen states [31, 35, 38]. Furthermore, they are not language specific; they can be implemented for various programming languages without too many alterations.

Current data-driven approaches have some limitations. Their hints often only inform students what to do without further explanation, which may cause uncertainty or confusion, and students not following up on the hints. Marwan et al. [27] added textual explanations to next-step hints. This feedback had a higher follow-up rate and interpretability score, and students perceived hints as more relevant. Unfortunately, these explanations had to be handcrafted. Additionally, Rivers found that novices wanted more elaborate feedback than experienced students [38]. Price et al. [36] identify situations where data-driven hints perform poorly overall, such as when students have code that diverges from standard solutions. Finally, even though data-driven approaches are considered less time-consuming, they often require complex transformations and comparisons. These shortcomings illustrate the possibility of improving existing data-driven techniques.

2.2 Large language models

Large language models are deep-learning models, trained on enormous amounts of data to generate output when given a task. A *prompt* is a request, question or instruction for what a user wants the model to generate. OpenAI's GPT-3, released in May 2020, is a breakthrough LLM [3]. Models such as Alpha-Code, codeBERT, and OpenAI's Codex focus on code synthesis. The human-like conversations and easy accessibility have made OpenAI's ChatGPT and GPT-4 immensely popular. GPT-4 outperforms other existing LLMs on several academic benchmarks [30].

2.2.1 LLM in programming education. LLMs may have significant impact on programming education [33]. They perform well on CS1/CS2 programming tasks [7, 10, 11, 33], which may cause over-reliance of novice programmers on LLMs. Furthermore, LLMs may produce syntactically incorrect code or contain constructs that are too complex and not appropriate for beginners [8].

Using an LLM may interfere with the problem-solving process. Students' focus may shift from solving the assignment to how to get an LLM to do what they want [34]. More experienced students using LLMs while coding often have difficulties understanding, editing, and debugging code snippets created by Copilot [41]. Students with access to Codex perform slightly better overall on a post-test than a group without access [17]. However, students with access make significantly more errors in coding tasks where they have to write code from scratch, possibly indicating an over-reliance.

LLMs also offer opportunities to support students and teachers in programming education. They can generate programming assignments, clarify programming error messages, generate code explanations, and help solving bugs [22, 24, 25, 39, 42].

There are a few studies on giving feedback using LLMs. Codex can create feedback on syntax errors consisting of a fixed program with a corresponding explanation [32]. The feedback LLMs give on incorrect submissions is sometimes useful, but often contains misleading information [20]. The feedback generated by an LLM for students' help requests when coding in a programming course often addresses at least one issue, but struggles to find all [14]. Although the LLM was asked not to include a model solution, it often did. Our work complements these studies by focusing on generating next-step programming hints.

3 METHOD

To answer our research questions we create a dataset with sequences of steps students take towards solving a programming problem (3.1), use these sequences to engineer a prompt for generating next-step hints (3.2), build the StAP-tutor, and evaluate the results (3.3). Figure 1 shows the workflow of this process.

3.1 Student program dataset

To create a dataset of sequences of steps novice students take when solving a programming problem, we used the dataset from Lyulina et al. [23], which contains snapshots of student programs for various tasks, collected with their *TaskTracker-tool*. This tool saves a code snapshot with every keystroke. The dataset consists of programs from 148 participants with various ages and experience, and solutions in multiple languages. We kept Python programs from participants with less than half a year of programming experience. We selected two exercises that required applying the most operations and programming constructs: Pies and Brackets, see Table 1.

We processed the data sequences as follows:

(1) *Remove duplicate program states.*
(2) *Remove program states with syntax errors.* We want to generate next-step hints, and not fix syntactically incorrect code.
(3) *Remove programs with incomplete steps.* We reduced the dataset by not considering single keystrokes; we only kept the last snapshot when a student was editing a line. We also removed snapshots in which students used print statements to trace their code and removed them shortly after.

3.2 Prompt engineering

To find out how best to instruct the LLM (OpenAI's gpt-3.5-turbo model) to generate the desired output, we performed iterative prompt engineering, where we determined how to proceed based on intermediate findings. We called the OpenAI API with various prompts and settings to generate the hints, see Section 4.

3.3 Evaluation

We built and integrated our findings into the StAP-tutor. The StAP-tutor is a web interface where students can practice their Python skills with the help of next-step hints. As teachers and students have different perspectives on effective feedback, we performed two evaluations to assess the quality of the generated hints: an evaluation by experts, and an experiment with students.

3.3.1 *Student evaluation.*

The Ethics and Privacy Quick Scan of Utrecht University classified this research as low-risk, with no fuller ethics review or privacy assessment required.

Participants. Using convenience sampling, we recruited three first-year Bachelor AI students at Utrecht University. They had some programming experience and were taking a Python course.

Exercise. The students worked on a different task than used for prompt engineering to validate if the final prompt generates good quality hints for other tasks too. The exercise "clumpCount" is taken from the CodingBat.org website, see Figure 1. It is typically solved in multiple steps, using loops and conditionals, making this task suitable for generating next-step hints. We adjusted the problem by explicitly describing the input and output requirements, similar to the tasks used for prompt engineering. We found that adding these complete descriptions helped with references to variables from the problem description and student code in the hints.

Experimental setup. The experiment lasted approximately one hour. We explicitly mentioned that the objective was to evaluate hints, not their programming experience. We suggested to ask for as many hints as they wanted, and to request hints when they started the next step. This probably deviates from a real-life situation, but we wanted to gather as many hints as possible. We assume this did not make a difference for the student rating of hints.

Students worked on the exercise for 45 minutes in the StAP-tutor. We asked them to rate each requested hint, as in MacNeil et al.'s experiment [24], where students immediately evaluate explanations after they are shown. After requesting a hint, the interface showed a pop-up with a rating request,see Figure 2. Students rate hints based on three statements: "The hint is clear", "The hint fits my work", and "The hint is helpful". We chose short questions with a 5-point Likert scale to make rating feedback easy and limit the time spent on rating. We also included the option to add a comment.

After working on the exercise, we asked the students several questions about how it went, how they would compare the tutor to ChatGPT, thoughts on improvements, and overall experience.

3.3.2 *Expert assessment.*

We conducted a qualitative evaluation of the hints by two experts: two authors of this paper with extensive experience as a TA and a lecturer, respectively. First, we composed a list of nine evaluation criteria based on the literature, see Table 2. For the *Feedback type*, we used Keuning et al.'s classification. Regarding phrasing, we evaluate the *Tone* and measure the *Length*. We do not consider longer hints to be better since we want to keep feedback informative but concise. The criteria *Personalised*, *Appropriate*, and *Misleading Information* directly correlate with the hint's effectiveness. Students prefer feedback linked to their work, which we capture in the criterion *Personalised* by checking references to the student's code or approach. We mark hints as *Appropriate*

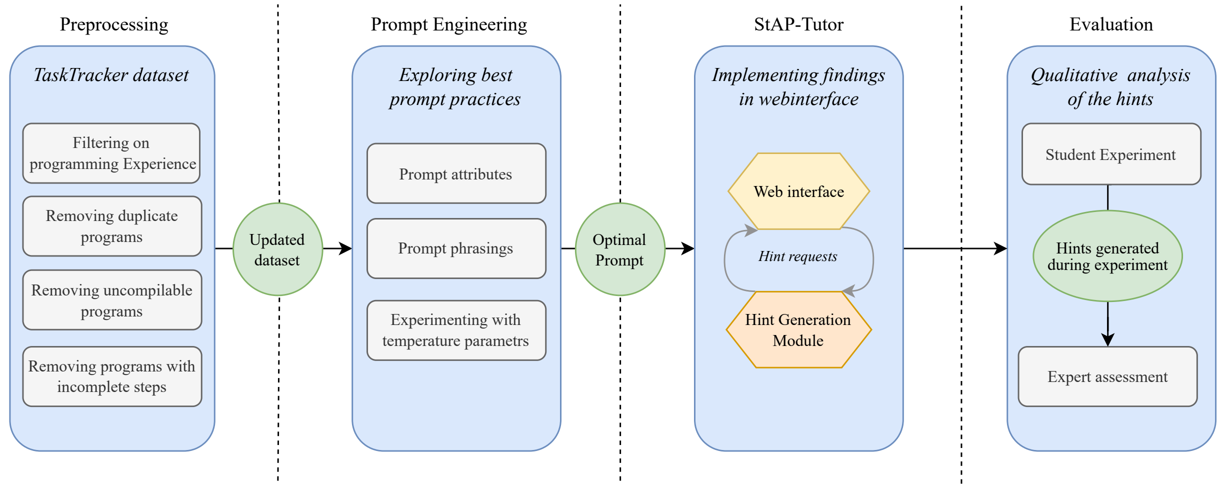

Preprocessing
TaskTracker dataset
Filtering on programming Experience
Removing duplicate programs
Removing uncompilable programs
Removing programs with incomplete steps
Updated dataset
Prompt Engineering
Exploring best prompt practices
Prompt attributes
Prompt phrasings
Experimenting with temperature parametrs
Optimal Prompt
StAP-Tutor
Implementing findings in webinterface
Web interface
Hint requests
Hint Generation Module
Evaluation
Qualitative analysis of the hints
Student Experiment
Hints generated during experiment
Expert assessment

Figure 1: Method of this work.

Table 1: Description of the exercises used for prompt engineering (pies and brackets) and the evaluation (clumps).

Pies	**Brackets**	**Clumps**
A single pie costs A dollars and B cents in the cafe. Calculate how many dollars and cents one needs to pay for N pies. *Input*: The program receives three numbers A - how many dollars a pie costs; B - how many cents a pie costs; N - how many pies do you need to buy *Output*: Print out two numbers: the cost of N pies in dollars and cents.	Place opening and closing brackets into the input string like this: for odd length: example → e(x(a(m)p)l)e; for even length: card → c(ar)d, but not c(a()r)d. *Input*: The program receives a string of English letters (lowercase and uppercase). *Output*: Print out the string with the brackets added.	Say that a "clump" in an array is a series of 2 or more adjacent elements of the same value. Return the number of clumps in the given array. For example, an array with the numbers [2,2,3,5,6,6,2] has 2 clumps. *Input*: The program receives a number n, followed by n lines with one integer per line. *Output*: Print out the number of clumps

when it fits the current program state. *Misleading information* are incorrect statements, which can result in misconceptions.

To verify our interpretations of the categories, we randomly selected 19 hints to compare. Table 3 shows the agreements and the corresponding inter-rater reliability with Cohen's Kappa. According to Landis et al.'s interpretation [21], *Level-of-detail* had a 'fair' and *Tone* a 'moderate' score. Because these are more subjective categories, this was not unexpected. Overall, there is substantial agreement, since we disagreed on only 13 of 152 values. We discussed differences, reached a consensus, and one author rated the remaining 29 hints.

4 PROMPT ENGINEERING

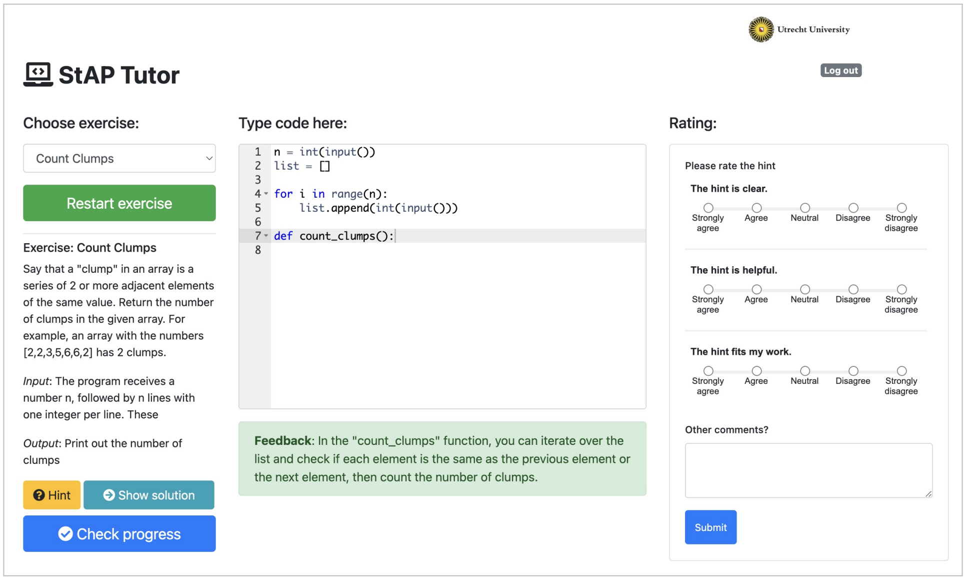
We performed prompt engineering in an iterative process, where we determined how to proceed based on intermediate findings. In this section, we discuss the results of our design process.

4.1 Prompt phrasing and attributes

We started with an exploratory phase in which we tried many different prompts for different programs. From our preprocessed dataset, we selected snapshots from three students working on both

Table 2: Evaluation Criteria.

Criteria	**Definition**
Feedback type	What type of feedback is the generated hint?
Information	Does the hint contain additional information, such as an explanation, tip or compliment?
Level-of-detail	Is the hint a bottom-out hint or a high-level description?
Personalised	Does the hint refer to the student's code or approach?
Appropriate	Is the hint a suitable next step, given the current state of the student program?
Specific	Is the hint limited to only one next step?
Misleading information	Does the hint contain misleading information?
Tone	Is the hint phrasing direct, neutral or friendly?
Length	What is the length of the hint in sentences?



Exercise: Count Clumps

Say that a "clump" in an array is a series of 2 or more adjacent elements of the same value. Return the number of clumps in the given array. For example, an array with the numbers [2,2,3,5,6,6,2] has 2 clumps.

Input: The program receives a number n, followed by n lines with one integer per line. These

Output: Print out the number of clumps

```
n = int(input())
list = []

for i in range(n):
    list.append(int(input()))

def count_clumps():
```

Feedback: In the "count_clumps" function, you can iterate over the list and check if each element is the same as the previous element or the next element, then count the number of clumps.

Please rate the hint

The hint is clear.
○ Strongly agree ○ Agree ○ Neutral ○ Disagree ○ Strongly disagree

The hint is helpful.
○ Strongly agree ○ Agree ○ Neutral ○ Disagree ○ Strongly disagree

The hint fits my work.
○ Strongly agree ○ Agree ○ Neutral ○ Disagree ○ Strongly disagree

Other comments?

Figure 2: Interaction with the StAP-tutor: the student is asked to rate the hint.

Table 3: Inter-rater reliability for the expert evaluation.

	Feedback Type	Information	Level-of-detail	Personalized	Appropriate	Specific	Misleading Info	Tone
Agreements	19/19	18/19	16/19	18/19	18/19	18/19	17/19	15/19
Cohen's kappa	0.872	0.787	0.387	0.883	0.855	0.642	0.683	0.467

assignments. These students used different approaches to solve the exercise, and had different programming competencies. We selected between 5 and 13 representative snapshots per student sequence.

The prompt consisted amongst others of the source code for which we wanted to generate a next-step hint, and an instruction describing the goal of the prompt. We first experimented with adding the problem description and model solution attributes to the prompt. Inspired by Sarsa et al. [39], we used formatting to denote the different components of the prompt. Fig. 3 shows an example. We phrased the prompt instruction in three different ways. We included the word "step" to specify the essence of what should be generated from the prompt. We created minor variations to examine how adding different keywords such as "student" or "hint" would affect the generated hints:

i) "What is the next step?"
ii) "Give a hint for the next step."
iii) "Explain the next step for a student".

We combined these 3 instructions with 4 possible combinations of the problem description and model solution attributes: a prompt with no attributes, one of the two and both. This resulted in $3 \cdot 4 = 12$ prompts. For each prompt we generated approximately 60 hints; 10 for each of the 3 students for 2 assignments. We set the temperature to 0.1 to limit the 'creativity' of the model and produce somewhat consistent results. The first author of this paper examined all results.

--Description--
Place opening and closing brackets into the input string like this: for odd length: example → e(x(a(m)p)l)e; for even length: card → c(ar)d, but not c(a()r)d.
The program receives a string of English letters (lowercase and uppercase) as an input.
Print out the string with the brackets added.

--Student Input--
v1 = input()

--Instruction--
Give a hint for the next step.

Output: Next, you can use an if statement to check the length of the input string to determine if it is odd or even.

Figure 3: Example prompt with output.

4.1.1 *Problem description and model solution.*

We found that only including the problem description resulted in the best outputs. A prompt without the problem description and model solution led to the model guessing what the student could do next, for example:

> "*The next step likely involves performing some mathematical operation using the values assigned to v2, v3, and v4.*"

At the start of the exercise, these hints could serve as inspiration on what to do next. However, as the student continued, the model was unable to generate feedback with useful suggestions.

Adding one of the two attributes caused the next-step hints to be more related to the student code and assignment. However, prompts with a model solution often produced hints that advised students to compare their code to the model solution. For instance:

> "*Think about how the student's code is different from the model solution. What changes need to be made to the student's code to make it work correctly?*"

Despite explicitly asking the model not to refer to the model solution, the model kept doing this. Also, the model would always suggest a step towards the provided model solution, even if a student was solving the problem in another (allowed and correct) way. For instance, an obvious solution for solving the brackets exercise is using a loop. One student who used join operations instead received several 'tips' to implement a loop. Or, for the pies exercises, we found students who did not compute the total costs in cents, in contrast to the model solution. As a result, the students received:

> "*The student's code seems to be attempting to calculate the cost of buying pies, but it is not following the same approach as the model solution. Try suggesting that the student should use the same formula as the model solution to calculate the total cost of buying pies.*"

Already using different variable names would lead to suggestions about changing those names. We tried to rephrase "Model solution" as "Example solution", however, this did not solve the problem. We decided to use prompts with only a problem description.

4.1.2 Instructions. We experimented with including different instructions in the prompt. We started with three different instruction phrasings (i–iii), shown in Table 4 with corresponding hints for an example student program. Although the instructions had only slight variations, the generated feedback had notably different results, especially regarding additional information and phrasing. The result of the prompt "What is the next step?" most frequently contained code, and the phrasing was a bit more straightforward and blunt. Consequently, we considered this prompt less suitable.

The prompt "Explain the next step for a student" resulted in relatively longer hints, while sometimes also explaining the student's code. Although explanations can help students, we only want to generate hints explaining how to proceed. The phrasing of the generated feedback with this prompt was similar to "Give a hint for the next step". Both yielded more friendly and carefully phrased hints. Furthermore, this instruction produced feedback formulated as a hint without revealing the exact answer, as in "The next step involves computing the total costs in cents."

Based on these observations, we combine the keywords from the last two prompts, "hint" and "student", to design new instructions. We omit "explain" as it caused long feedback and undesirable code explanations. When trying the prompt: "Give this student a hint for the next step", we found it often generated long outputs. So, we replaced it with two variations that explicitly instruct the model to create shorter hints. Our new set of instructions were:

(ii) "Give a hint for the next step."
(iv) "Give this student a short hint for the next step."
(v) "Give this student a hint for the next step. The hint should be one or two sentences."

4.1.3 Temperature. The next step focused on analysing the impact of the *temperature* parameter. We investigated values from 0.1 to 0.9, with intermediate increments of 0.2 for instruction (ii). We briefly compared prompts with only a description and a description combined with a model solution. We wanted to rule out that increasing the temperature would overcome the issues mentioned in section 4.1.1. Indeed, we still encountered the same problems and thus chose to continue only with the description prompt.

As the temperature value increased, we noted that hints contained more unusual or unexpected suggestions. For example, recommendations for using a stack in the brackets exercise, which is unnecessary and probably an unknown data structure to novices. However, we saw improvements too, where hints were more tailored to the students code. We compared the number of useful and unhelpful suggestions. We noticed a shift from having more disadvantages than advantages for every prompt around a temperature of 0.7. Hence, we picked a temperature of 0.5 for our final prompts.

4.1.4 Choosing the best prompt. To choose the best prompt, we used the three prompts from the previous section with a temperature value of 0.5. We randomly picked ten student programs for both exercises. Then, two authors compared the hints generated for each program and ranked them from 1 (best) to 3 (worst). The final score for each prompt was the sum of its total ratings, as shown in table 5. We selected the prompt with the best total score (v).

4.2 Best prompt practices

We conclude this section by reflecting on **SQ1**: What prompt characteristics are suitable for generating effective next-step hints with LLMs? We found that providing more information, such as a model solution, doesn't always improve the output. LLMs give hints that relate strongly to such a solution, while teachers would be able to recognise alternative solutions and help students without forcing them to change their approach. A related issue was that LLMs often explicitly refer to the model solution, even when instructed not to do so. Hellas et al. [14] noticed this problem as well. When we included the sentence "..use at most three sentences" in the prompt, the output would have *exactly* three sentences. We believe that LLMs cannot handle or interpret certain constraints very well.

The presentation of feedback is important; we found that keywords such as "hint" and "student" often help with personal references, explanations, a friendly tone, and compliments. For example:

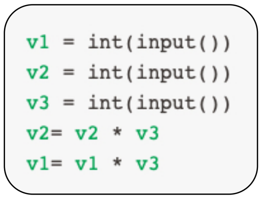
> "*The formula for calculating the cost of N pies is correct, but you need to separate the result into dollars and cents. Remember that 100 cents make 1 dollar. You can use the modulo operator (%) to get the remainder when dividing by 100, which gives the cent value.*".

Additionally, using the keyword "hint" prevented output that gave away the answer, as we saw with prompts "What is the next step?", which often produced code, while "Give a hint for the next step" gave tips such as "the next step involves using a for a loop."

The instruction should be carefully formulated. For example, the model did not recognise when a student completed the assignment and would still recommend a 'next step'. We briefly experimented

Table 4: Example hints for the code on the left using different prompt instructions.

	Prompt	Hint


`v1 = int(input())`
`v2 = int(input())`
`v3 = int(input())`
`v2= v2 * v3`
`v1= v1 * v3`
	What is the next step	The next step is to calculate the total cost of N pies by adding v1 and v2.

	Give a hint for the next step	The next step should involve calculating the total cost of N pies using the updated values of v1 and v2, and then converting the total cost into dollars and cents.
	Explain the next step for a student	It seems like you are trying to calculate the total cost of N pies. You have correctly multiplied the cents by the number of pies, but you also need to multiply the dollars by the number of pies. You can do this by updating the value of v1 to be v1 multiplied by v3. Then you can use the updated values of v1 and v2 to calculate the total cost of N pies.

Table 5: Prompt ranking.

	Prompt (ii)	Prompt (iv)	Prompt (v)
Brackets	20	27	**16**
Pies	25	19	**17**
Total score	45	46	**33**

with adding something to the prompt to overcome this problem, such as "If the student is done, give a compliment. Else, give a next step hint." The hints for this prompt would recognise situations where the model was incorrect before. However, the feedback generated was much longer and had (too) many compliments. Adding "give a compliment" changed every output where we had expected this to happen for only some of the hints.

Finally, increasing the temperature value might help produce better feedback. However, this also increases the risk of unusual or useless suggestions, which can be harmful, especially for novices. More experienced programmers could probably get inspiration from such feedback and recognise unhelpful hints. Increasing the temperature also resulted in more references to the student's code and approach. Choosing this value depends on the context and should be determined through experimentation.

5 EVALUATING NEXT-STEP HINTS

5.1 StAP-tutor

After discovering and choosing the best prompts, we incorporated our findings in our StAP-tutor (**St**ep **A**ssisted **P**rogramming Tutor). The StAP-tutor is a web interface combined with a hint-generation module. The interface has various functionalities, such as choosing exercises, requesting hints and checking your solution, as shown in Fig. 2. After starting an assignment, the student can ask for help by pressing the hint button. This button sends a request to the hint-generating module, an API written in Python. This API constructs a prompt for the current student program and calls the OpenAI API with the **gpt-3.5-turbo** model. Then, the API sends the hint back to the interface, where it is displayed to the student.

5.2 Student experiment

In the experiment, the three students requested 11, 20 and 17 hints, in total 48. The student's hint ratings are in Figure 4. We notice they often marked the hints as clear and suitable for their work.

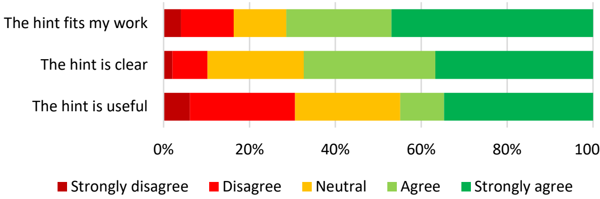


Figure 4: Student hint ratings (n=48).

However, they were less convinced about the usefulness of the feedback. Only about half of the time, they tagged the hints as helpful. Some students included a comment with an explanation. One reason was that the feedback was useful, as it was the same as at the beginning of the exercise, but the student had already made significant progress. Another student commented that the hint contradicted a previously given hint, suggesting that the work done did not contribute to the solution.

Even though all students stated that the exercise was aligned with their current skills and programming level, no student finished with a correct solution. They particularly struggled with the logic for correctly counting the clumps, which required tracking and implementing when to 'count' a clump with conditional statements.

After the experiment, we asked the students about their experiences and opinions. Overall, they had a positive impression of the tutor and appreciated the hints, especially at the start of the assignment. At the start, the feedback fitted their work and had practical suggestions for what to do next. However, after a while, when the student was stuck with more complex code, the hints were not helpful anymore. The students needed more detail or practical suggestions, but the feedback was too vague.

All students mentioned requesting multiple hints for the same code. As the LLM is not deterministic, it might help to regenerate a hint. Students reported that regenerating hints returned a different phrasing or provides more or new information. However, one of the students stated that returning similar hints for the same code could even be more helpful if those hints entail more detailed content.

Finally, the students agreed that compared to ChatGPT, the StAP-tutor focuses more on your existing code instead of providing a general solution with a lot of additional information. They thought this might help with learning, as this also reduces the temptation

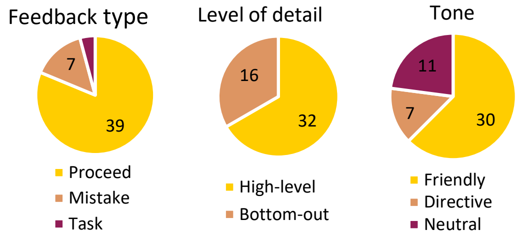


Figure 5: Hint characteristics.

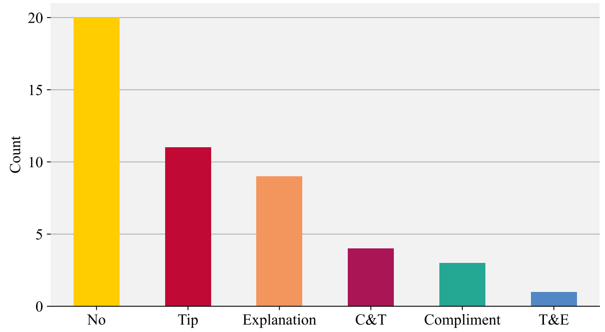


Figure 6: Counts for additional information. C&T = compliment and tip, T&E = tip with explanation.

to copy and paste the assignment and generate a whole solution. However, as one student pointed out, an advantage of ChatGPT is that there is a possibility to ask more specific questions yourself.

5.3 Expert assessment

All 48 generated hints were classified by one or two experts. Figure 5 shows the frequencies for the criteria *Feedback Type*, *Level-of-detail* and *Tone*. The majority of the generated hints correspond with the type *Knowledge About How To Proceed*. We found that the hints are more often high-level than bottom-out. Note, in contrast with the type of feedback, we did not state the desired level of detail in the prompt. Finally, we found that the feedback mostly had a friendly tone. We interpreted the tone as friendly when the hints were more suggestive (*e.g.,* “Consider trying to change the condition of the for loop”) than directive (*e.g.,* “Change the for loop condition”).

We found that friendly hints often contain additional information. Figure 6 shows the frequencies for every additional information category. We occasionally noticed that feedback contained smaller tips in addition to the ‘main’ hint, which often referred to something the student was not yet working on. For example, “Also, try to think about the conditions under which a clump exists.” or “Don’t forget to handle the cases where the clump ends and a new clump begins”.

All evaluation metrics with binary values are presented in Figure 7. Feedback is frequently personalised, appropriate and specific. We saw hints with explicit references to variables from the student’s code and suggestions referring to their current implementation. Furthermore, the feedback usually entailed only one specific step, which was appropriate regarding the student’s progress. However, we should be cautious about hints containing misleading information. We noted the misleading information was not notably wrong at first sight, but often more subtle. For example, feedback explained incorrectly why some piece of code is incorrect: “The for loop of your current code may reach an out-of-range index error if it reaches the end of the list without finding a clump”. Or, it gives suggestions that will not help to solve the problem: “You need to make sure your program doesn’t break when the input array has only one element.” Finally, we found misleading information is often connected to hints that refer to edge cases, while a correct solution did not require handling these cases separately.

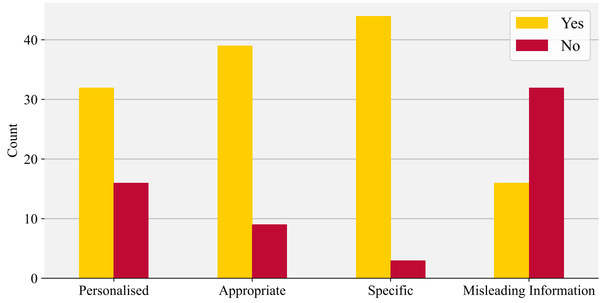


Figure 7: Frequencies of binary evaluation metrics.

5.4 Conclusion

In this section we answer **SQ2**: What are students’ and experts’ perceptions of the quality of LLM-generated next-step hints, and how do they rate them? We found that both students and experts are positive. While the feedback provided concise and tailored support, it was not always helpful and sometimes contained misleading information. Solving programming exercises entails breaking down a problem into smaller sub-tasks, and next-step hints may help students performing a sequence of steps leading to a solution.

Overall, the students found the hints clear. Most feedback consisted of 1 or 2 sentences, with some outliers to 3. We think LLM-generated feedback from these prompts has the right level of complexity and length. LLMs respond with appropriate and potentially helpful answers, as in the work of Hellas et al. [14]. Students sometimes ask for hints multiple times for the same code repeatedly.

All students indicated they needed more support than the generated next-step hints. The prompt we used generated mainly high-level hints. These hints may help the students get started and support them with constructing an idea for an overall approach. Yet, when approaching the end of the exercise, students want more detailed feedback related to their solution.

The feedback frequently included additional information, such as compliments, tips and explanations. Students liked the ‘motivational words’, which made the interaction feel more personal. We found it important that the feedback was not overly confident and directive, to encourage a student to think critically about the hint.

An obvious limitation is that hints occasionally contain misleading information. This issue is hard to resolve with further prompt engineering as we believe it is more related to the overall capability of LLMs themselves. We expect that as the LLM performance increases, the amount of misleading information will decrease. When using LLMs for teaching, we should be cautious and warn students about them providing incorrect information.

6 DISCUSSION

This section reviews our main research question and discusses the limitations regarding prompt-engineering, the experimental set-up, and reproducibility, and how follow-up research may address these.

6.1 Generating next-step hints using LLMs

Answering **RQ1**, "To what extent can we use LLMs to generate informative and effective next-step hints for Python introductory exercises?", we found it is best to use prompts that refer to a problem description and include keywords related to the application context, such as “student” or “hint”. Compared to other data-driven approaches, LLMs can generate personalised, tailored feedback, while they do not require large datasets or complex computational methods. In contrast with standard hints, which often are mere code edit suggestions, these hints contain compliments, explanations and tips. This additional information was well-received by the students. Nevertheless, there are still some points for improvement, as LLM-generated feedback may entail misleading information.

6.2 Threats to validity

6.2.1 Prompt engineering. Although we extensively experimented with engineering prompts, we had to limit ourselves. First, we primarily examined the effects of using different attributes, phrasings and temperature values isolated from each other. This approach allowed us to analyse the influence of these characteristics independently. These factors may influence each other, but investigating all possible combinations would require too much time. Second, we only analysed a small set of prompt instructions. As we were content with the preliminary results, we chose not to explore different phrasings, which might have led to increased hints quality. Lastly, we did not use strict evaluation criteria in the prompt engineering phase. Because we wanted to get a general impression of how the LLM performed for a wide variety of code states, we did not generate multiple hints for one student program.

6.2.2 Experimental setup. We performed the experiments with only three students. Furthermore, we only examined their opinions instead of factors like learning gain. Therefore, we cannot make strong statements about the effect of the hints.

In our expert assessment, we tried to capture most important factors for effective feedback in our evaluation criteria. Nevertheless, capturing effectiveness based on a few characteristics is challenging since this depends on many factors. For instance, students could ask for hints for themselves, which is not necessarily proven to be the best practice [16]. Implementing other methods for timing the feedback, however, require a method for automating hint delivery, which was outside the scope of this work.

6.2.3 Reproducibility. A known issue with doing LLM-studies is that models are constantly updated, which might produce other results. During our experiments, OpenAI released GPT-4. We performed some minor experiments using GPT-4 and compared its results with GPT-3.5-turbo. We found that the same prompt for both models outputs, at first glance, different results. GPT-4 created hints with good suggestions we had not seen before with our best prompts. We expect our prompt practices also work for updated versions, but other research is required to confirm this.

6.3 Future work

Although not perfect, at this time, LLM-generated hints certainly have potential. We propose various avenues for future work.

Students may benefit from help at different levels depending on the context. Once students are stuck, they might require more in-depth feedback that offers better guidance than high-level hints. Our StAP-tutor uses only one prompt for hint generation, which generates both bottom-out and high-level hints. We could further investigate the relation between prompts and the level of detail of the hints they generate. These findings could be combined with advanced methods for student modelling to provide personalised feedback and guidance based on a student’s progress. For instance, depending on the number of hints requested for a specific state, a different prompt could be used to obtain more guided feedback.

Students pointed out that they could not influence the hint topic. Prather et al. found similar student frustrations while working with Copilot [34]. Hints occasionally did not correspond to the part of the code students were struggling with. For example, the StAP-tutor suggested writing already-written pieces of code. We propose to investigate student control, so students can indicate what part of the code they want help with, and to instruct the model to give more specific feedback.

Other ideas are adding few-shot learning or experimenting with giving the prompt multiple solutions. With few-shot-learning, examples are included in the prompts [3], from which the LLMs could learn and follow its structure. However, this requires a set of expert-validated example hints, which we did not have for our dataset.

7 CONCLUSION

We investigated how to use LLMs to generate next-step hints for introductory programming exercises in Python. These hints should support students in their learning, and not give away the entire solution. First, we explored various prompt practices to discover what prompts would yield the best feedback. We analysed the effects of adding a model solution and description to the prompt, using different instructions and varying the temperature. We found it is best to only include a problem description and use keywords such as “student” and “hint” in the prompt. In addition, increasing the temperature parameter might contribute to more personalised feedback. However, too high values may cause unhelpful suggestions.

We then created the StAP-tutor based on these results. With this tool, students can ask for hints while doing programming exercises. Students working with the tutor had an overall positive impression of the hints. The LLM-generated feedback was personalised, appropriate and contained helpful additional information such as tips and explanations. Our expert assessment support these findings, but also found that the hints contained misleading information. Another issue was that the high-level feedback did not support students at every stage of their problem-solving. Our study shows the potential of using LLMs in programming education by generating hints, and proposes several areas for future work.

REFERENCES

[1] John R Anderson and Brian J Reiser. 1985. The LISP tutor. *Byte* 10, 4 (1985).

[2] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard-Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proc. of SIGCSE*. 500–506.

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

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[5] Phillip Dawson, Michael Henderson, Paige Mahoney, Michael Phillips, Tracii Ryan, David Boud, and Elizabeth Molloy. 2019. What makes for effective feedback: Staff and student perspectives. *Assess. & Eval. in Higher Ed.* 44, 1 (2019), 25–36.

[6] Galina Deeva, Daria Bogdanova, Estefanía Serral, Monique Snoeck, and Jochen De Weerdt. 2021. A review of automated feedback systems for learners: Classification framework, challenges and opportunities. *Comp. & Ed.* (2021).

[7] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proc. of SIGCSE*. 1136–1142.

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

[9] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. 2022. Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic? *arXiv e-prints* (2022).

[10] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Proc. of ACE*. 10–19.

[11] 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. In *Proc. of ACE*. 97–104.

[12] Qiang Hao, David H Smith IV, Lu Ding, Amy Ko, Camille Ottaway, Jack Wilson, Kai H Arakawa, Alistair Turcan, Timothy Poehlman, and Tyler Greer. 2022. Towards understanding the effective design of automated formative feedback for programming assignments. *Computer Science Education* 32, 1 (2022), 105–127.

[13] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of educational research* 77, 1 (2007), 81–112.

[14] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. In *Proc. of ICER*. 93–105.

[15] Alastair Irons and Sam Elkington. 2021. *Enhancing learning through formative assessment and feedback*. Routledge.

[16] Johan Jeuring, Hieke Keuning, Samiha Marwan, Dennis Bouvier, Cruz Izu, Natalie Kiesler, Teemu Lehtinen, Dominic Lohr, Andrew Peterson, and Sami Sarsa. 2022. Towards Giving Timely Formative Feedback and Hints to Novice Programmers. In *ITiCSE Working Group Reports*. 95–115.

[17] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proc. of CHI*.

[18] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2014. Strategy-based feedback in a programming tutor. In *Proc. of CSERC*. 43–54.

[19] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM TOCE* 19, 1 (2018), 1–43.

[20] Natalie Kiesler, Dominic Lohr, and Hieke Keuning. 2023. Exploring the Potential of Large Language Models to Generate Formative Programming Feedback. *arXiv preprint arXiv:2309.00029* (2023).

[21] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* (1977), 159–174.

[22] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using large language models to enhance programming error messages. In *Proc. of SIGCSE*. 563–569.

[23] Elena Lyulina, Anastasiia Birillo, Vladimir Kovalenko, and Timofey Bryksin. 2021. TaskTracker-Tool: A Toolkit for Tracking of Code Snapshots and Activity Data During Solution of Programming Tasks. In *Proc. of SIGCSE*. 495–501.

[24] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proc. of SIGCSE*. 931–937.

[25] 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. In *Proc. of ICER*. 37–39.

[26] Yana Malysheva and Caitlin Kelleher. 2022. An Algorithm for Generating Explainable Corrections to Student Code. In *Proc. of Koli Calling*. 1–11.

[27] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. The impact of adding textual explanations to next-step hints in a novice programming environment. In *Proc. of ITiCSE*. 520–526.

[28] Elham Mousavinasab, Nahid Zarifsanaiey, Sharareh R. Niakan Kalhori, Mahnaz Rakhshan, Leila Keikha, and Marjan Ghazi Saeedi. 2021. Intelligent tutoring systems: a systematic review of characteristics, applications, and evaluation methods. *Interactive Learning Environments* 29, 1 (2021), 142–163.

[29] Susanne Narciss. 2008. Feedback strategies for interactive learning tasks. In *Handbook of research on educ. communications and technology*. 125–143.

[30] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023).

[31] Benjamin Paassen, Barbara Hammer, Thomas W Price, Tiffany Barnes, Sebastian Gross, Niels Pinkwart, et al. 2018. The Continuous Hint Factory-Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *JEDM* 10, 1 (2018).

[32] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. *arXiv preprint arXiv:2302.04662* (2023).

[33] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Peterson, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots are Here: Navigating the Generative AI Revolution in Computing Education. *arXiv preprint arXiv:2310.00658* (2023).

[34] 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. *ACM Trans. Comput.-Hum. Interact.* (2023).

[35] Thomas W Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating data-driven hints for open-ended programming. *Int. Educ. Data Mining Society* (2016).

[36] Thomas W Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A comparison of the quality of data-driven programming hint generation algorithms. *International Journal of Artificial Intelligence in Education* 29 (2019), 368–395.

[37] Thomas W Price, Zhongxiu Liu, Veronica Cateté, and Tiffany Barnes. 2017. Factors influencing students' help-seeking behavior while programming with human and computer tutors. In *Proc. of ICER*. 127–135.

[38] Kelly Rivers and Kenneth Koedinger. 2017. Data-driven hint generation in vast solution spaces. *Int. J. of Artificial Intelligence in Education* 27 (2017), 37–64.

[39] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proc. of ICER*. 27–43.

[40] Valerie J Shute. 2008. Focus on formative feedback. *Rev. of ed. res.* 78, 1 (2008).

[41] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI conference extended abstracts*. 1–7.

[42] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing Bugs in Python Assignments Using Large Language Models. *arXiv preprint arXiv:2209.14876* (2022).