Automated Next-Step Hint Generation For Introductory Programming Using Large Language Models

Lianne Roest

Department of Information and Computing Sciences

Utrecht University

First Supervisor
Hieke Keuning

Second Supervisor
Johan Jeuring

A thesis submitted for the MSc degree in

Computing Science

July 2023

Acknowledgements

After devoting nine months to this project, I look back on it with a sense of fulfilment and appreciation. First, I would like to express my gratitude to Dr. Hieke Keuning for supervising this thesis. Her guidance and feedback helped me to shape this work in a rapidly growing research field. Our meetings were always enjoyable, and her insights on various aspects of the topic were incredibly valuable. I would also like to thank Prof. Dr. Johan Jeuring and our study group for our sessions where I could share the progress of my work and receive helpful suggestions, and be inspired by other students working on similar projects. I would like to thank Nienke, Kim, Tessa, Merve and Xavi for our countless Dalton study sessions, they were an absolute pleasure. I also wish to thank my family, who have always supported me. Lastly, I want to express my sincerest appreciation to Stan for his help, belief and encouragement I received throughout this project. To everyone mentioned, thank you for being part of this experience. I am truly grateful, forever and always.

Abstract

With recent advances in their performances, large language models (LLMs) are now more popular than ever. LLMs, such as ChatGPT, possess various skills, such as answering questions, writing essays or solving programming exercises. Recently, these models have become easily accessible, and researchers have expressed concerns regarding their impact on programming education. This work explores how LLMs can contribute to programming education by supporting students with automated next-step hints generated by LLMs. We investigate prompt practices that lead to effective next-step hints and use these insights to build our StAP-tutor. Furthermore, we evaluate our developed tool by conducting a student experiment 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. Hence, this work demonstrates the potential for LLM-generated feedback, but further research is required to explore its practical implementation.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Background

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 exercises done by students to facilitate their learning process [22]. There exists a lot of variation in the type of feedback and how tools generate feedback. Feedback may include hints for code edits, references to relevant concepts, or suggestions for reading material. One specific category considered in this work is next-step hints. These hints focus on helping students how to proceed when they are stuck while solving programming assignments. For instance, the ITAP-tutor suggests: "In line 1 replace 'saturday' with 'Saturday' in the right side of the comparison operation" [52].

As mentioned before, feedback can be automatically generated with different techniques. Generating next-step hints, specifically, is often associated with data-driven methods [50]. Data-driven approaches use data from students instead of predefined models. These models, used by more traditional methods, are manually designed and precisely define what, when and how feedback is provided to the students. Data-driven algorithms use student data instead and, for example, compare incomplete student solutions to other student programs to extract next-step hints [52]. These hints can then guide students closer to the correct solution.

Some issues with data-driven approaches are that they often require a large amount of data. Furthermore, more substantial limitations are that most hints are exact directions for what the student should do, revealing the answer for the student and interfering with learning opportunities. The hints only contain instructions on what to do and lack any elaborations, as in the provided example before. Work from Price et al. showed that the lack of explanations makes next-step hints difficult for novices to interpret and connect to their current code [51]. Hints without explanations have a decreased following rate while adding explanations can improve students' understanding of the hints and perceived relevance. In addition, it can benefit the overall learning process and the student's trust in the system [35]. Yet, these proposed approaches require a lot of manual labour, with a handcrafted explanation per situation, nullifying the advantage of data-driven approaches regarding developing time

In this work, we apply state-of-the-art data-driven language models to develop next-step hints, i.e. large language models (LLMs). In the last few years, there has been a significant increase in the development and, especially, in the performance of LLMs. LLMs are trained on enormous amounts of data and have shown to be very good at specific tasks, such as generating text, images, and now also code [5]. Especially with the arrival of the models from OpenAI (GPT-3, Codex, and GPT-4), LLMs have recently received much attention, both in and outside the academic world. Several studies show that Copilot and Codex perform relatively well at solving introductory programming assignments [9, 14]. Moreover, as these models are also available for students, researchers express their concerns regarding plagiarism, integrity, and learning in the programming education field [14, 3].

Novice programmers may be particularly susceptible to the 'dangers' of using LLMs as their pair programmers. Using large language models might not benefit the student's learning process, as it prevents the student from working on the assignment themselves. In addition, novice programmers do not have the skills yet to understand code and recognise code of good quality. Therefore, students might not know how to design effective prompts to get the desired results [11]. And finally, students might overestimate their coding abilities using such LLMs [44], which can cause incorrect reflection on their programming performance and possibly, more mistakes

With this study, we aim to give a positive turn to using LLMs in programming education. We have built a tool that employs LLMs to generate next-step hints. By letting LLMs create the feedback, we do not need to design our models and avoid the complex tasks that data-driven methods often require, such as gathering data or training such a model. We generate next-step hints since our approach matches a data-driven approach, and data-driven algorithms often produce next-step hints [50]. Data-driven feedback generation algorithms often solely produce edit-based hints, which are precise instructions on what to do. With our approach, we hope to enhance the provided feedback with additional information, such as explanations. Furthermore, our tool allows students to use the potential and help of LLMs but in a controlled way, without exposing them to too much information and choices regarding promoting engineering. So far, other related research on LLMs in programming education investigates the LLMs' performance or generates supporting material using LLMs, such as code explanations [33, 32] or programming exercises [53]. Many of these studies do not report anything on the process of their prompt designing. We found two studies related to the topic of generating feedback. One study concerns generating hints for syntax error corrections [45], and the other explores generating fixes for student code by help requests [18]. Yet no work exists on using LLMs to generate specifically next-step (semantic) hints.

1.2 Research questions

Our main objective is to examine how we can use LLMs to generate high-quality next-step hints. In this work, we specifically focus on introductory Python assignments, as Python is a programming language often used for novice programming courses. We will experiment with prompt engineering practices to generate next-step hints. We will use our findings to build our tool, the StAP-tutor (**St**ep **A**ssisted **P**rogramming tutor). Finally, we will evaluate the quality of the generated hints with an experiment and expert assessment. To this end, we formulated the following main research question:

> **RQ1** To what extent can we use LLMs to generate informative and effective next-step hints for Python introductory exercises?

Furthermore, we will analyse two sub-questions. Many studies on applying LLMs in programming education do not report how they designed their prompts, the decisions made during prompt engineering and why they made them. We will investigate different prompts to get an overview of how one designs prompts for generating next-step hints. We aim to create next-step feedback that is more than just suggestive edits, as often generated with data-driven methods.

> **RQ2** What prompt characteristics are suitable for generating effective next-step hints with LLMs?

Finally, we evaluate the quality of the LLM-generated next-step hints. We perform an evaluation both by experts and by conducting student experiments. Students and teachers do not necessarily agree on what feedback or hints are effective or helpful. With two different evaluation methods, we consider both to obtain a global idea of the quality of the generated hints.

> **RQ3** What are students' and experts' perceptions of the quality of LLM-generated next-step hints, and how do they rate them?

The remainder of this work is structured as follows. First, Chapters 2 and 3 discuss related work regarding providing effective feedback, generating automated feedback and using LLMs in programming education. Then, we present a short overview of this work's method in Chapter 4. After this, we have a more in-depth discussion on our methods regarding the prompt-engineering and hint generation with corresponding results in Chapters 5 and 6, respectively. Chapter 7 discusses the limitations of our work with possible directions for future research. Finally, In Chapter 8, we give our conclusions.

2 Automated feedback

Digital learning tools have been developed across multiple educational domains. Most focus on teaching in STEM, but others support learning languages, medicine or other skills. Nevertheless, a substantial part of the systems teaches topics within computer science and programming [8, 37]. Due to the ever-growing interest in programming, the demand for teaching programming courses on a large scale has increased significantly. Hence, the interest in developing programming teaching tools has increased as well.

The main focus of the research related to educational programming tools is providing automated feedback [37]. We provide a short overview, first discussing effective feedback in general, and then programming feedback types and generation techniques. Aside from feedback, other example features from educational programming tools are personalised course material recommendations, adaptive testing or student evaluation. As our research is limited to providing automated hints, we centre the remainder of this section around feedback for programming.

2.1 Providing effective feedback

Feedback is considered essential in the process of learning [17]. Feedback can enhance learning by correcting misconceptions or mistakes. In addition, feedback can improve the students' motivation by providing positive comments or guidance when the student is stuck. All feedback that consists of information or learning activities that support and facilitate student learning is classified as formative feedback [19]. Its counterpart, summative feedback, consists of assessment activities, resulting in grades that evaluate the student's performance.

Various variables impact the effectiveness of formative feedback, such as its complexity [54]. Complexity relates to giving elaborate or concise feedback. Elaborated feedback helps students understand why something is wrong and can guide students in the right direction through explanations or tips. As a counterpart, if feedback is too long, many learners will not pay attention to it, rendering it useless. Another essential factor is timing. Feedback can be provided by intervening or waiting: until the student requests feedback themselves or finishes an assignment. Another variable of feedback is its nature, which can be positive or negative. In contrast with negative feedback, positive feedback reinforces what students are doing well, for example, to emphasise the student is correctly implementing assignment requirements. Both can, however, have beneficial effects on learning [17].

Although many studies on effective feedback exist, defining effective feedback remains complicated due to its many aspects. Besides, even students and teachers have different ideas about what feedback is effective. Dawson *et al.* performed a qualitative study of teachers' and students' ideas and opinions of effective feedback [7]. Although teachers and students agree that the overall purpose of feedback should improve student performance, they have other beliefs on what makes feedback effective. While teachers believe in the importance of timing, modalities and connected tasks, students prefer high-quality feedback comments that make feedback effective. These comments should be detailed and personalised to the student's work. Wisniewski *et al.* found that feedback is more effective when it contains more information [57], which could correspond to the student's preference for more detailed feedback. Their work shows that simple reinforcement and punishment have slight effects, while high-information feedback is the most helpful.

2.2 Feedback types

Keuning *et al.* conducted a systematic review of automated programming feedback in which they identify feedback categories, feedback generation and evaluation methods [22]. With their classifications, based on an earlier one created by Narciss [38], they find the following feedback categories:

- *Knowledge About Task Constraints*: This first category entails hints related to the task itself, the proper approach for the exercise or the fulfilment of the assignment's requirements.
- *Knowledge About Concepts*: Feedback that provides explanations on relevant knowledge concepts. This category also includes illustrating examples that can help the student's understanding.
- *Knowledge About Mistakes*: Feedback covering all hints related to mistakes. Examples are hints about errors, such as compiler errors. Mistakes regarding the solution and style issues are also part of this category.
- *Knowledge About How to Proceed*: This feedback relates to guiding the student on what to do next. The hints belonging to this group can be in different forms, such as suggestions, questions or instructions. The goal of the hint can be to correct an error or guide the student to take the next step towards a correct solution.
- *Knowledge About Meta-cognition*: The last category is on a more abstract level of feedback and essentially covers insights into the student's knowledge about their performance and progress.

Keuning *et al.* found that most programming learning tool provides feedback with knowledge about mistakes by implementing automated testing, as this approach is generally easy to implement [22]. However, Hao *et al.* discovered that feedback of this type is not as effective compared to when more informative and detailed feedback is included [16]. Only presenting mistakes as hints requires students to interpret the results themselves, which may interfere with learning from their misconceptions.

This work focuses on generating hints for the fourth category: feedback on how to proceed. Next-step hints support can prevent students from getting stuck while working on their assignments, which is typically a problem for novice programmers. Other feedback types do not provide this kind of support.

2.3 Feedback techniques

Over the years, many techniques have been developed for generating automated feedback, with general approaches and approaches specific to the programming domain. Keuning *et al.* identified three universal techniques for generating feedback and five domain-specific to the field of programming

General feedback techniques:

- *Model tracing*: Tools that implement model tracing give immediate feedback at each step during the problem-solving process [36]. Model tracing requires a definition for production rules corresponding to correct steps, and buggy rules, representing typical mistakes.
- *Constraint-based modelling*: Constraint-based modelling identifies errors in student solutions given a set of constraints [36]. The technique checks if the solution state satisfies all requirements. If not, based on the violated constraints, a tool implementing this approach can provide predefined feedback belonging to those constraints.
- *Data analysis*: Data-driven approaches use large sets of student solutions from the past to generate hints.

The first two approaches require a significant amount of manual labour. Every knowledge component, production rule or constraint has to be defined by experts. Especially in the domain of programming, where assignments often have different solutions, semantically or syntactically, this can be a very time-consuming task. Data analysis approaches overcome these issues by employing algorithms by learning patterns or strategies that lead to correct solutions from data instead of predefining these strategies. More regarding data-driven approaches will be discussed in section 2.4.

Programming-specific feedback techniques:

- *Dynamic code analysis*: Dynamic code analysis: Dynamic code analysis produces feedback using automated testing and gives knowledge about mistakes. Generally, automated testing compares the program output on several test cases to the expected output. Other techniques for automatic testing are unit testing and property-based testing.

- *Basic static analysis*: With basic static analysis, a tool provides feedback without running the student program. The most rudimentary feedback by static analysis is syntactical errors generated by a compiler. In addition, static analysis can check student programs on misunderstood concepts and other solution requirements, such as required code structures [55].
- *Program transformations*: Program transformations can facilitate automated feedback generation. Program transformations help simplify programs to make it easier to compare multiple programs. For example, to compare a student program to a model solution. Examples of program transformations are normalisation or generating abstract syntax trees (ASTs).
- *Intention-based diagnosis*: With intention-based diagnosis, a tool tries to identify the strategy of the student [26]. Intentions are defined as goals or plans and implemented similarly to the rules from constrained-based modelling.
- *External tools*: Keuning *et al.* defined this category as tools that use other tools that had not the same creator and are more advanced than standard compilers or static analysis tools.

The choice of feedback technique determines, to some extent, the type of feedback generated. For example, with model tracing, the core of the domain model is focused on production rules which encapsulate progress. Therefore, feedback generated with model tracing is often associated with knowledge of how to proceed, as this is also the core of the related domain model. Similarly, knowledge of mistakes is often the result of automated testing. A substantial part of the data-driven approaches is also related to generating hints associated with knowledge of how to proceed,*i.e.,* next-step hints. [50]. In general, data-driven methods for programming operate as follows: first, it compares the current student state (from the moment of the hint request) to a desired target state, often a correct solution. Then, an algorithm identifies the differences or required edits to transfer from the current state to the target state. From there, the algorithm extracts one or more next steps [34, 52].

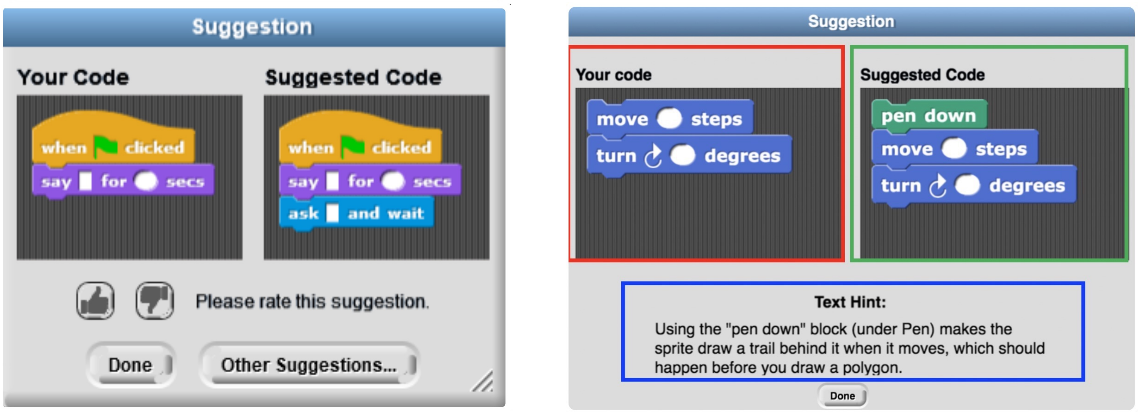
2.4 Data-driven next-step hints

During the last few years, a growing number of studies showed interest in developing data-driven approaches for generating feedback. As data-driven methods avoid needing experts to create a complex domain model, they are considered less time-consuming than traditional feedback techniques.

Barnes *et al.* were one of the first researchers who used student data in their study for automated hints [1]. Their approach used a Markov Decision Process to generate hints for logic proofs. Data-driven approaches have since then also been applied within the programming education domain. Note that although studies label themselves as data-driven approaches, their actual implementation can vary in multiple ways. For example, they may use different algorithms, methods for code comparisons etc.

However, some studies share similarities in their approaches, for instance, using abstract syntax trees for their state-space representation (ASTs) [49, 34, 52]. In other words, they used ASTs to represent source codes as states so they could easily compare students' codes and create suggestions on how to proceed. Rivers *et al.* developed the ITAP tutor that followed this approach. By transforming student programs to ASTs and comparing the current state to other student data, their algorithm constructs a path of edits to a correct solution [52]. From this path of potential modifications, the algorithm can extract a hint and presents it to the student to guide them in the right direction. All hints follow the following format: [Location info] + [action verb 1] + [old val] + [action verb 2] + [new val] + [context]. Applying this format produces hints as: "In line 1 replace 'saturday' with 'Sunday' in the right side of the comparison operation."

Price *et al.* conducted several studies on data-driven next-step hints for their iSnap environment, an extension of the Snap*!* environment, based on Scratch, a block-based programming language [49, 48], shown in figure 2.1. Their algorithm generates a set of multiple edits the student could apply per request, again using ASTs. Other implementations of data-driven approaches for next-step hints have other underlying techniques as their base, such as program synthesis [25] and automated testing [39]. For example, Obermuller *et al.* use automated tests to decide which solutions from the program pool are candidates for hint generation.



Text Hint:

Using the "pen down" block (under Pen) makes the sprite draw a trail behind it when it moves, which should happen before you draw a polygon.

Figure 2.1: Examples of next-step hints from the ISnap environment [49, 35]

Note that although data-driven approaches are often associated with next-step hints, other methods may also utilize the possibilities with historical data. For instance, Fossati *et al.* researched the effects of feedback with positive and negative nature in a tutoring system for linked lists. They employ historical student data to decide what feedback was well-suited to the current state of the student [15].

As previously mentioned in section 2.3, data-driven methods are less time-consuming to create than other feedback generation techniques. Another advantage is they can generate hints for never-before-seen states, making them more applicable than traditional methods [52, 48, 42]. Furthermore, they are not language specific. Consequently, they can be implemented for various programming languages without too many alterations.

There are still some limitations to the current data-driven approaches. First, their hints often only inform students what to do without further explanation or additional information. This may cause uncertainty or confusion for students, with a chance of students not following the given hints. This hypothesis was confirmed by Marwan *et al.* [35]. In their study, the authors added textual explanations to their next-step hints, from which an example is shown in figure 2.1. They found that when provided with explanations, the feedback had a higher follow rate and interpretability score, and students had an improved perception of its relevance. Unfortunately, these explanations had to be crafted by hand, annulling the benefits of data-driven approaches compared to expert-driven ones. Additionally, Rivers found that novice programmers wanted more elaborate feedback compared to experienced students [52]. Data-driven approaches currently do not provide hints with details.

Second, Price *et al.* identify situations where data-driven hints perform poorly overall, such as when students have code that diverges from obvious or standard solutions [50]. Finally, even though data-driven approaches are considered less time-consuming than traditional methods, they often remain very complex due to transformations and comparing student programs. These shortcomings illustrate the possibility of improving existing data-driven techniques.

3 Large language models

Very recent advances in AI, especially in the last decade, have enhanced the powers of large language models. Large language models (LLMs) are deep-learning models. These models are trained on enormous amounts of data and can generate output given a specific task with the knowledge they were trained on. LLMs exist for different modalities, such as generating natural language, images or code, often depending on the type of data the model is trained with.

LLMs generate output given so-called prompts. A prompt defines a request, question or instruction for what the user wants the model to generate. For example, when creating an image, the user formulates the prompt: "Create a painting from a student sitting on the beach but in a science fiction world". During their training process, LLMs learn the probabilities of token sequences and set the values from the parameters in their model accordingly. Once trained and presented with a prompt, the model outputs the sequence of tokens with the highest overall probability. This output is the image or text that the user asked to generate. Note that the output LLMs generate are non-deterministic as LLMs often include some randomness or noise in their predictions. These are incorporated since randomness can sometimes yield better and more creative results.

Over the years, several LLMs have been created and published, some specifically designed for code synthesis. Examples of such models are Alpha-Code [29], codeBERT [12], and OpenAI's Codex [41], which is a model built upon the GPT-3 model by OpenAI. OpenAI and its large language models have received much attention recently because of the release of ChatGPT and GPT-4. With its human-like conversations and easy accessibility, especially ChatGPT has been immensively popular.

3.1 The models of OpenAI

OpenAI created several LLMs over the last years. GPT-3, released in May 2020, is one of the first breakthrough models, with more than 10 times the parameters compared to other language models from that time [4]. The first testing with GPT-3 focused on language-related tasks and demonstrates that human evaluators had trouble distinguishing GPT-3 generated news articles from human-written texts [4].

OpenAI's Codex is a fine-tuned model of the famous GPT-3 model. Its training data contains an enormous amount of source code from publicly available sources, including code in public GitHub repositories [41]. As a result, Codex can also produce source code in addition to the 'standard' capabilities of its predecessor GPT-3 [5]. The initial research on Codex shows that the fine-tuned model has improved performance on code synthesis for a dataset with human-written programming problems [5]. In addition, other studies confirmed the range of Codex's capabilities, such as code repair [47] and code documentation [58]. Since March 2023, the API for Codex is not in production anymore. Yet, GitHub Copilot, an IDE plugin created by GitHub and OpenAI, still utilises Codex to give live code suggestions in IDEs.

In November 2022, ChatGPT was released, which made human-like conversations with LLMs possible. In March 2023, GPT-4 followed and now outperforms other existing LLMs on several academic benchmarks with, among others, NLP and coding tasks [40].

The easiest way to access one of the models from OpenAI is via ChatGPT's web interface. Other than that, one can use GitHub's Copilot or access the models via APIs available on the OpenAI platform. Some models, such as ChatGPT-plus and GPT-4, are only accessible with paid subscriptions. The main difference between these tools is that, for the API, much more refined settings are possible. One can choose between the different base models of GPT-3 along with other more advanced parameter settings, such as temperature, custom stopping sequences and the maximum number of tokens generated per request. The temperature parameter determines the level of 'creativity'. The higher the temperature, the more diverse the generated answers are.

Naturally, the more settings and options available for the user to experiment with, responses can also be increasingly divergent. Doderlein et al. find that with the right parameter settings, the performance of Codex can significantly improve[11]. Moreover, their study also shows a strong dependency on the optimal temperature and prompt variation. The results indicate that this may also vary per programming language. Altogether, these findings demonstrate the trade-off between user efforts and model performance and the need for careful consideration when one uses these models.

3.2 LLM in programming education

Since the arrival of LLMs, multiple researchers have expressed their concerns about the negative impact these models might have on programming education [14]. Recent work shows that Codex performed as well as the top 25 % of the students in an introductory programming course [14] and in a CS2 course on data structures and algorithms [13]. In addition, other work suggests that Copilot can solve 60% of a set of beginner Python assignments on the first try and solve 70% of the remaining exercises with adjustments and refinements of the initial prompts given to Copilot [9].

In their paper publication of GPT-3, Chen et al. acknowledge that novice programmers may be particularly susceptible to overreliance on LLMs [5]. Besides learning 'to code', novice programmers need to develop skills, such as computational thinking, since programming requires abstract, higher-order thinking and reflection. Relying too heavily on code generation tools may hinder the development of such skills [10]. Furthermore, ChatGPT and similar models produce syntactically incorrect code or contain constructs that are too complex and not appropriate for beginners [10].

Using LLM may also interfere with the problem-solving process of students. Students' focus may wander to how to get a LLM to do what the student wants instead of keeping attention to solving the assignment [46]. Vaithilingam et al. studied more experienced programming students using LLMs while coding [56]. They reported that students often spent too much time reading the large code blocks suggested by Copilot and frequently switching between thinking, reading, and debugging. However, novice programmers probably spend even more time comprehending generated code and get more easily confused. In their experiment, novice students had difficulties understanding, editing, and debugging code snippets created by Copilot. As mentioned earlier, this hinders the student's solving process and may interfere with learning opportunities.

In a study comparing two groups of students from which only one had access to Codex, Kazemitbaar et al. found that students with access performed slightly better overall on a taken post-test. However, students with access made significantly more errors in coding tasks where students had to write code from scratch, possibly indicating an overreliance [21]. In a similar comparison study, Perry et al. discovered that participants without access to an LLM had written more secure code, while users with access believed they had written more secure code [44].

These and similar findings are the base for worries about students' integrity when they have such a model accessible. Hence, these findings may be overwhelming and perhaps overshadow all possibilities LLMs can bring to programming education. Fortunately, several studies already illustrate these opportunities and how LLMs can support students and teachers in programming education.

3.2.1 Creating supporting material

As a first example, Sarsa et al. let Codex generate programming assignments based on existing exercises [53]. They provide Codex with prompts containing example exercises as inputs. The authors use four base assignments with corresponding model solutions and unit tests, contextual and technical keywords for the topic and learning subjects of the assignments. They found that generated exercises by Codex were overall reasonable and had, most of the time, correct corresponding solutions. When the assignments, solutions or test cases contained some mistakes, the authors could often fix these mistakes. For example, they tried regenerating the content by presenting the model the prompt. Moreover, even the incorrect content could still inspire teachers, thus reducing the workload.

3.2.2 Code explanations

Besides clarifying programming error messages, several studies research generating explanations for source code [53, 33, 32]. The ability to trace and explain code is an essential part of learning to program and has a strong relationship with the ability to write code [30]. Generating code explanations could assist students in learning to trace their code and eventually explain their code on their own.

Sarsa et al. include generating explanations in their paper with the generation of programming exercises but did a substantially smaller analysis on this subject [53]. They found that Codex's explanation often contains small mistakes related to incorrect or missing conditional branches or invalid comparisons. The explorative study conducted by MacNeil et al. resulted in a classification of the explanation types Codex generates [33]. They identify eight distinctive types, among others, code summary, listing relevant programming concepts and tracing the execution of code. Unfortunately, there was no report on prompt inputs or the quality of the explanations, as the authors intend to evaluate this in their future research.

Another study investigated incorporating code explanations generated by Codex and GPT-3 in a web software development E-book [32]. Remarkably, the authors found that GPT-3 produced higher-quality outputs than Codex. Codex tends to go off-topic and often includes source code, while a requirement is that all explanations should be in plain English. Another interesting finding was that adding examples to the prompt was not beneficial for the quality of the explanations as the model overfitted the structure of the shots. The study's experiment shows that students consider the explanations relevant and helpful for their learning process. Furthermore, LLM-created code explanations are significantly easier to understand and contain more accurate code summaries compared to student-generated ones [27]. This work points out how LLM-generated explanations potentially can help students trace their code, as they can be produced automatically on demand.

3.2.3 Providing student assistance

After writing their code, students should be able to fix the bugs in their program. Zhang et al. examined if Codex could solve students' bugs in Python programs [59]. They formalise a method for employing Codex, which first corrects the syntactical errors and then repairs the semantic ones. During the process, they generated multiple corrections and applied the one having the smallest token-edit distance to the student's program. Their method could repair 96.5 % of the student programs. Compared to the baseline model, constructed by combining multiple state-of-the-art Python repair engines, the approach outperforms the baseline with a lower token-edit distance.

Such fixes have the potential to help students when transforming them into feedback, which is something Phung et al. did with their research [45]. With Codex, they create feedback for syntax errors consisting of a fixed program with a corresponding natural language explanation. Furthermore, they implement a run-time validation mechanism which successfully determines whether the generated output is appropriate for sharing with the student.

Another related study from Leinonen et al. uses Codex to clarify error messages [28]. Understanding programming error messages is an essential part of programming, yet, it is also a difficult task, especially for novice programmers. The authors produce additional explanations of the error messages with Codex, which clarify the reason for the error and provide instructions on how to fix the error. With qualitative analysis, they found that 70% of Codex's explanations were valid. However, these results would differ with a temperature parameter set to 0. Experimenting with this parameter showed that other values would result in a significant decrease in performance, again demonstrating Codex's performance's strong dependence on parameter settings.

Hellas et al. collected code from students' help requests during an online programming course and presented it with a prompt to an LLM, asking for suggestions on how to fix it [18]. They found that GPT-3.5 outperforms Codex in most aspects. While both LLMs often can find at least one issue, neither excels at finding them all. Interestingly, they note that although the LLM is ordered not to, their responses frequently contain a model solution. With supporting students as the main objective, this is naturally strictly undesired outcome, preventing the students from creating a solution themselves.

The discussed studies show that LLMs may positively impact programming education, with still many opportunities on how LLMs can contribute unexplored. This work will hopefully join and complement these studies by generating next-step programming hints.

4 Method

Our method for hint generation with LLMs consists of four components; data preprocessing, prompt engineering, building the StAP-tutor and evaluation. The workflow of this process is shown in Figure 4.1. We first needed a dataset of student programs for which we could generate hints and experiment with prompt engineering. Section 4.1 discusses the process of creating a suitable dataset. We used the dataset from Lyulina *et al.*, which contains snapshots of student programs for various exercises. We downsized this dataset such that differences between consecutive entries were less fine-grained. We used two assignments from the dataset, Pies and Brackets, for analysing different prompt strategies for generating next-step hints. We investigated the effect of including model solutions and problem descriptions and used different phrasings and temperature values, described in section 4.2.

After completing prompt engineering, we used the best practices we found to develop our StAP-tutor. The StAP-tutor is a web interface where students can practice their Python skills with the help of next-step hints. Finally, we conducted an experiment where students worked on a new exercise and rated the feedback they received. We then evaluated the generated hints with two experts and compared the results of those ratings with the ratings from the students. We discuss our evaluation methods in section 4.3.

In the rest of this chapter, we discuss the steps of our method in more detail, starting with the preprocessing of the *TaskTracker-tool* dataset. Then we provide a overview of the phases of prompt engineering and our evaluation methods.

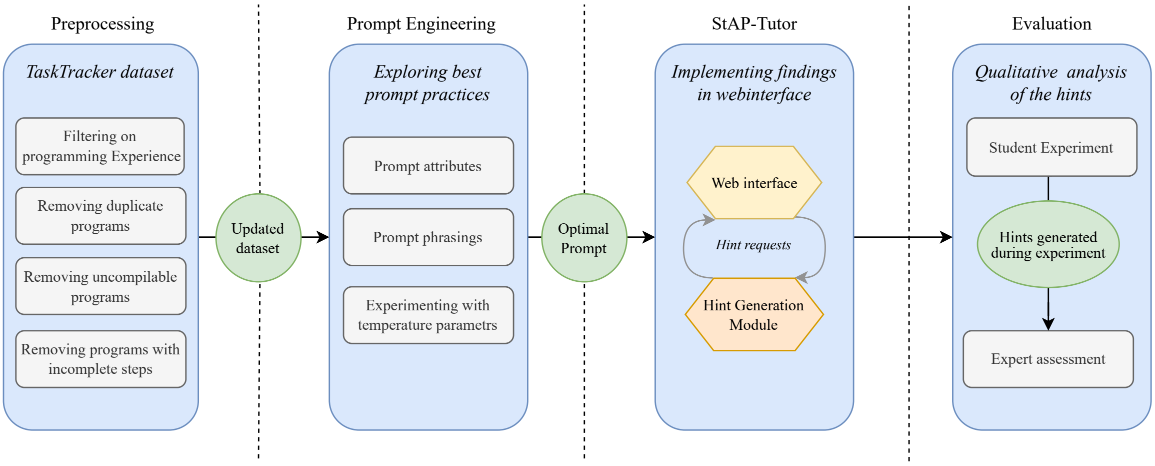


Figure 4.1: Method of this work

4.1 Student program dataset

Lyulina *et al.*. [31] provide a dataset with programs collected with their developed *TaskTracker-tool.* After participants install the IDE plugin, the *TaskTracker-tool*collects consecutive code changes and IDE actions while they work on a programming task. The program is captured every time it is altered by the participant. Thus, with every keystroke, a snapshot of the participant's code is saved to the dataset, which makes the dataset quite fine-grained. For each assignment, the dataset consists of a CSV file for each participant, which consists of rows of consecutive program snapshots.

The dataset consists of programs from 148 participants aged from 11 to 40. Their programming experiences range from zero to more than six years of programming. The researchers accepted solutions in languages Python, Java, Kotlin or C++. For this work, we only considered the Python solutions and programs from participants with programming experience of less than a half year, as we focused on generating hints for introductory programming exercises. We selected two of the six exercises that required applying and implementing the most operations and programming constructs: Pies and Brackets. Table 4.1 shows the description of the assignments, with an example test case and solution.

We performed data processing as we did not aim to give feedback or hints when students were in the mid of writing a word or completing a step. In this work, we consider a line as a completed step, for example, implementing the condition of a while-loop. We wanted to generate hints for a set of student programs as diverse as possible. We performed several operations on the dataset from the *TaskTracker-tool*to obtain our new dataset where consecutive programs have more substantial steps in between. Figure 4.2 shows how we deleted programs with incomplete lines. We merged programs that belong to the same "state", i.e., programs where the student was working on the same step. We performed the following data processing operations:

1. *Removing duplicate programs*: We started by removing duplicate programs, as duplicate snapshots belong to the same program state.
2. *Removing uncompilable source code*: We then filtered on source code with syntax errors to remove programs that immediately caused errors. With this step, we also removed programs that contained unfinished keywords. This step was necessary as we aim to provide help for the next steps and not fix syntactically incorrect programs.
3. *Removing programs with incomplete steps*: Finally, we removed programs where students worked on one specific line for a sequence of successive programs, so a number line in the IDE, not a statement. We consider these programs to contain unfinished step(s). From such a sequence of programs, we selected the last snapshot for our dataset. We observed that students sometimes used print statements to trace their code and shortly removed them after. When this happened, we chose not to include

Table 4.1: Description of the exercises used for prompt engineering

Name	**Pies**	**Brackets**
Description	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 as an input 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) as an input. *Output*: Print out the string with the brackets added.

Example solution

Pies

```python
A = int(input())
B = int(input())
N = int(input())
price_per_pie = (A * 100) + B
total_cost = price_per_pie * N
dollars = total_cost // 100
cents = total_cost % 100
print(dollars, cents)
```

Brackets

```python
v = input()
n = len(v)
result = ””
for i in range(0, n//2 - 1):
    result += v[i] + ”(”

if n % 2 == 0:
    result += v[n//2 - 1]
    result += v[n//2]
else:
    result += v[n//2 - 1] + ”(”
    result += v[n//2]

for i in range(n//2 + 1, n):
        result += ”)” + v[i]

print(result)
```

Name	**Pies**	**Brackets**
Example testcase	Input: 2 50 4 Output: 10	Input: example Output: e(x(a(m)p)l)e

those snapshots in the dataset, as the print statements do not contribute to the program's functionality. In Figure 4.2, we see how we reduced a sequence of programs to one snapshot in the final dataset. The first row represents a complete state for line 4. After completing this step, the student proceeds with line 5 (the snapshots with the blue outlines). We see that after three snapshots, line 5 is complete. However, the student added and removed a print statement (red outline), from which we inferred that it was merely included for debugging purposes. We chose the snapshot after removing the print statement as the final state for line 5 and selected this snapshot in our final dataset.

```
1 v1 = str(input())
2 v2 = len(v1)
3 v3 = []
4 for v4 in v1:
```

```
1 v1 = str(input())
2 v2 = len(v1)
3 v3 = []
4 for v4 in v1:
5     v2
```

```
1 v1 = str(input())
2 v2 = len(v1)
3 v3 = []
4 for v4 in v1:
5     v3.append()
```

```
1 v1 = str(input())
2 v2 = len(v1)
3 v3 = []
4 for v4 in v1:
5     v3.append(v4)
```

```
1 v1 = str(input())
2 v2 = len(v1)
3 v3 = []
4 for v4 in v1:
5     v3.append(v4)
6     print(v4)
```

```
1 v1 = str(input())
2 v2 = len(v1)
3 v3 = []
4 for v4 in v1:
5     v3.append(v4)
```

```
1 v1 = str(input())
2 v2 = len(v1)
3 v3 = []
4 for v4 in v1:
5     v3.append(v4)
```

Figure 4.2: Process of removing incomplete steps

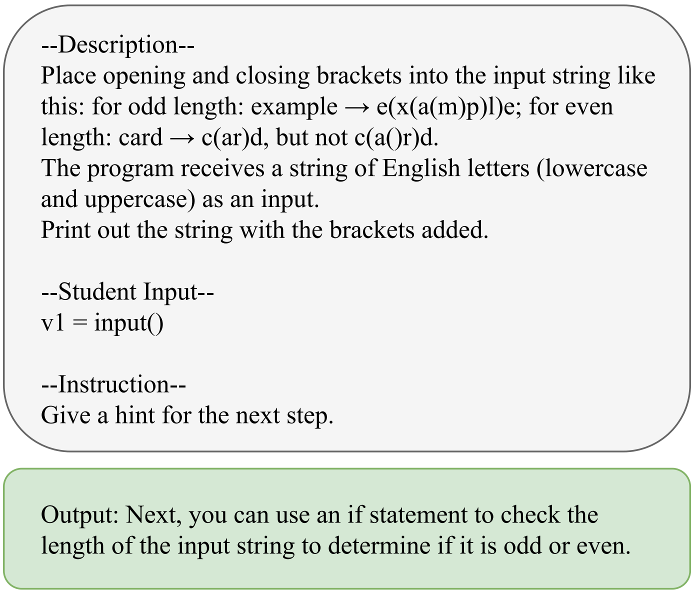
Note that performing these operations does not necessarily result in perfect series of code construction with line-by-line built programs. Firstly, students do not work linearly on their assignments. They jump back and forward between lines, can start with a high-level approach etc. In addition, in the final dataset, it is possible to have some jumps between two consecutive programs. For example, when a student writes syntactically correct code after finishing two steps, two successive programs have an addition of two lines of code instead of one.

However, with our constructed dataset, we now have a set of programs that models overall the step-wise progress of programming by students. In the next section, we explain how we used these programs to design our prompts.

4.2 Prompt engineering

Our main priority was to get a general idea of how we should instruct the LLM to generate our desired output. For our project, we used the **gpt-3.5-turbo** API. Whilst working on this project, GTP-4 was released. However, we could not incorporate this model into this work within the given timeframe. Instead of immediately starting with a strict criteria-based evaluation, we explored many different prompts for different students. We used a subset from our obtained dataset after processing. As mentioned before, we used two exercises: Pies and Brackets. As the goal is to explore how the LLM behaves, we wanted a diverse student set possible. From our preprocessed dataset, we selected three students for both assignments. We sought students that varied in their approaches to the exercise and ability to complete the assignment. We generated hints for two intermediate steps to limit the number in the first iteration, which resulted in a range of 5 to 13 programs per student.

The prompt always consisted of a student program, the source code for which we aimed to generate the next-step hint. We first experimented with adding two attributes to the prompt: problem description and model solution. We used the formatting of the prompt to denote the different aspects of the prompt. Sarsa *et al.*. [53] noted that denoting parts of their prompt yields better results. Finally, the prompt always had an instruction in which we encapsulated the goal of the prompt. We show an example prompt in Figure 4.3.



--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 4.3: Example prompt with output

Furthermore, we started with three different phrasings for the instruction part. We formulated these prompts and assumed that including the word “step” would specify the essence of what the prompt should output. We decided to create minor variations to examine how adding different keywords such as “student” or “hint” would affect the generated hints. These premises led to the following three instructions:

1. “What is the next step?”
2. “Give a hint for the next step.”
3. “Explain the next step for a student”.

We then combined these instructions with four possible combinations of attributes to a prompt: a prompt with no attributes, one of the two and both. Altogether, we created $3 \cdot 4 = 12$ different prompts for our initial round of analysing prompts. We generated approximately 60 hints with every prompt, 10 for each of the three students for both assignments. We initially set the temperature for this first iteration of prompt engineering to 0.1, as we wanted to limit the “creativity” of the language model and produce somewhat consistent results to begin with.

The next step in prompt engineering focused on analysing the impact of the temperature parameter on the generated hints. We investigated values from 0.1 to 0.9, with intermediate increments of 0.2. This part was completed individually by the author of this work. We examined all results and registered our impressions, findings and striking discoveries. For this part of prompt engineering, we chose not to perform a strict evaluation based on criteria. We preferred this approach as it allows for viewing a much higher amount of hints and student programs. Furthermore, we did not know what to expect beforehand, and this method let us familiarise ourselves with how adjustments to prompts will affect the produced hint.

In the next chapter 5, we discuss this process in more detail, accompanied by intermediate results. After completing the prompt engineering, we built and integrated our findings into the StAP-tutor. In the following section, we discuss how we evaluated our tool.

4.3 Evaluation

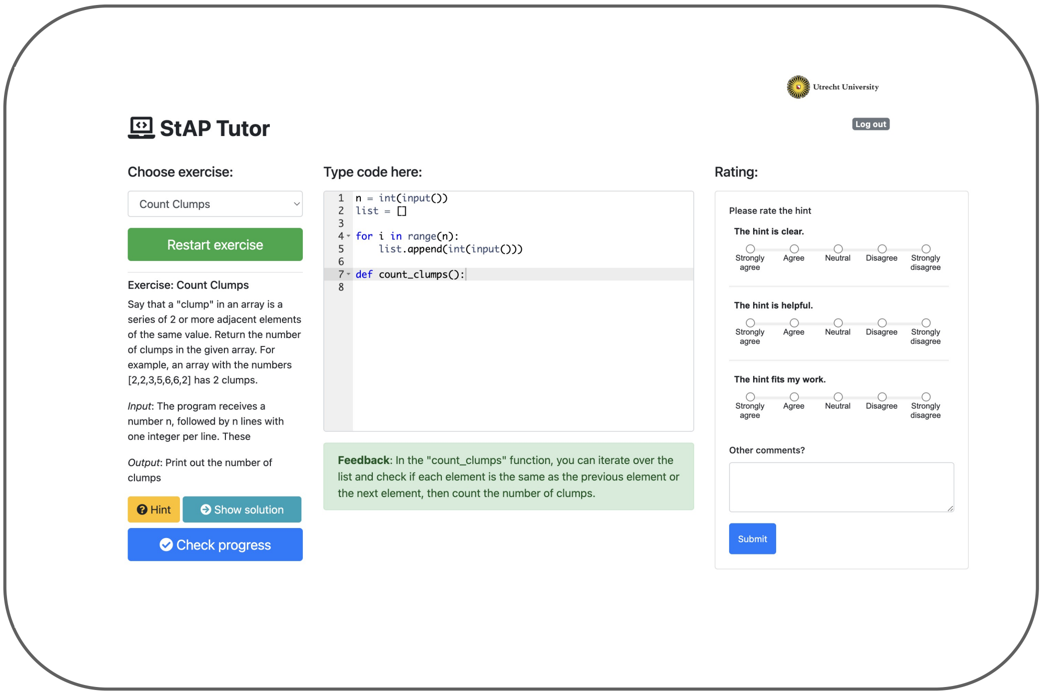
After building the StAP-tutor, we used it for two evaluations to assess the quality of the generated hints: an evaluation by experts, and an evaluation by students through an experiment. As described in section 2.1, teachers and students have different perspectives on effective feedback. We, therefore, gathered data from both sides to get a complete view of the hints’ quality. We first let students program freely work on an assignment with the StAP-tutor. During this experiment, the students were also asked to rate the hint, and so collecting a new dataset with the student programs with corresponding hints. We finally evaluated the hints from the experiment with two experts based on several criteria and compared the findings to the student’s opinions.

4.3.1 Student evaluation

We maintained an approach for the student experiment similar to the work of MacNeil et al. [32], where students immediately evaluate explanations after appearing. After requesting a hint, the interface immediately showed a pop-up with a rating request, as displayed in Figure 4.4. We let students rate hints based on three statements:

1. The hint is clear.
2. The hint fits my work.
3. The hint is helpful.

We chose short and straightforward questions with a 5-point Likert scale to limit the resistance to rate feedback and limit the time spent on answering. We specifically asked the students if the hints were clear and personalised, as we found in Chapter 2.1 that these properties are associated with effective feedback according to students. Asking the students about the helpfulness of the hints is self-evident. Finally, we included the option for the participants to add a comment.



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

Three students participated in the experiment. The students were recruited based on convenience sampling. Through former teaching and amicable relations, we found students willing to participate in the experiment. All are first-year students in the Bachelor of Artificial Intelligence at Utrecht University. This specific group of students already had some programming experience and were, at that time, following a course that taught Python. We considered experience in Python a prerequisite for our participants since the generated hints should support students in how to proceed instead of identifying and fixing syntactical errors.

Before carrying out the experiment, we evaluated our method with the Ethics and Privacy Quick Scan of the Utrecht University Research Institute of Information and Computing Sciences. They classified this research as low-risk, with no fuller ethics review or privacy assessment required.

Experimental setup

During the experiments, the students worked on an assignment individually in a private room. All students worked on a provided device (Macbook Air 2021). The total duration of the experiment was roughly one hour, from which the students worked on the assignments for 45 minutes. At the start of the experiment, we explained the goal of this research and the experiment. We explicitly mentioned that the objective was to evaluate the hints, not their programming experience. Then, we instructed the students on the experiment's workflow and assigned them a random number for saving data. We stimulated the students to ask for as many hints as they wanted. Next, we asked them to request hints when they thought they were starting the next step, which does not necessarily correspond to a real-life situation. However, we wanted to prevent students from holding back, and gather as many rated hints as possible. Furthermore, we assume this will not change the ratings of the students.

To begin with the experiment, the student logged in with their assigned number. We let the students work on a different assignment than used for prompt engineering. Hence, we could validate if the prompt generates quality hints for this assignment and generalises to other exercises. The exercise is called "clumpCount" and is publicly available on CodingBat.org [43]. Its solution contains loops and implementing logic with conditionals, consisting of several steps, making this exercise particularly suitable for generating next-step hints. We made some adjustments to the problem description. We wanted the assignment to explicitly describe the input and output requirements, similar to the exercises used for prompt engineering. We found that adding these complete descriptions helped with references to variables from the problem description and student code. Accordingly, the exercise now included reading input, which also serves as an extra step. The descriptions with and without adjustments can be found in table 4.2.

After 45 minutes of working on the exercise, we asked several final questions about the students' overall impressions. We asked how they thought it went, how they would compare using the StAP-tutor to using ChatGPT, thoughts on improvements and their overall experience of the StAP-tutor. After the experiments, we collected the data for our expert assessment, which we will discuss in the next section.

Table 4.2: CountClump problem descriptions

System	**Description**
CodingBat	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. countClumps([1, 2, 2, 3, 4, 4]) → 2 countClumps([1, 1, 2, 1, 1]) → 2 countClumps([1, 1, 1, 1, 1]) → 1
StAP-tutor	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
Example solution	

```
n = int(input())
lst = []
for j in range(0, n):
    lst.append(int(input()))

clump_count = 0
i = 0
clump = False
while i < n - 1:
    current = lst[i]
    if current == lst[i + 1]:
        if not clump:
            clump = True
            clump_count += 1
     else:
        clump = False
    i +=1

print(clump_count)
```

4.3.2 Expert assessment

After conducting the experiments, we performed a qualitative evaluation of the hints by a team of two experts. We created a list of evaluation criteria beforehand, shown in table 4.3. Before starting our evaluation, we assessed a randomly selected subset of six hints to compare and check if we had similar understanding and interpretations of the criteria and discussed until we had a final agreement. We adapted the classification of Keuning et al. for the first criterion, omitting the category Knowledge of Meta-cognition since we did not observe hints from this category during the prompt engineering phase. Furthermore, we incorporated the *Level-of-detail* and *Additional Information* provided with the feedback as criteria. Regarding the hints' phrasing, we evaluate the tone and measure the length in the number of sentences. We do not consider longer hints to be better since we want to prevent overloading the students with content, keeping the feedback informative but concise. We consider the criteria *Personalised*, *Appropriate*, and *Misleading Information* essential as they directly correlate with the hint's effectiveness. Students prefer feedback that is highly linked to their work. We encapsulate this in the criterion *Personalised*, for which we check references to the student's code or approach. We mark hints as *Appropriate* when it fits the current state of the student program. The content should connect to the student's progress so far. We view misleading information as incorrect statements, which can result in misconceptions among the student. For instance, hints can incorrectly state that the student accurately implemented a step. As a result, this could damage the students' understanding.

Table 4.3: 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 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 phrasing of the hint direct, neutral or friendly?
Length	What is the length of the hint in sentences?

Due to time constraints, one of the experts evaluated all other hints while the other assessed half of the remaining hints. To validate our rating process, we documented the number of agreements and calculated the inter-rater reliability per evaluation metric using Cohen's Kappa [6]. This metric expresses the agreement between two annotators on a classification problem. In our case, we classify the hints on their quality based on our criteria.

5 Prompt engineering

5.1 Engineering process

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

5.1.1 Problem description and solution

We first experimented with adding problem descriptions and model solutions to the prompt. We found that only including the problem description resulted in the best outputs. A prompt without a 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*".

The model attempted to provide a hint without addressing the overall purpose of the code or the assignment. At the start of the exercise, these hints could serve as inspiration for the student on what to do next. However, as the student continued with the assignment, the model was unable to generate feedback that could support the student in completing the exercise.

Adding one of the two attributes solved this problem and made the next-step hints more related to the student code and assignment. However, the prompts with a model solution often produce hints that advise 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?*"

Even after editing the prompts and explicitly commanding the model not to refer to the solution, the model kept doing this. Another issue when including a model solution was that model seemed incapable understand that the model solution is not strictly the only possible solution. Using other variable names than provided in the description generated hints concerning replacing the variable names. Or, the model would not recognise the student had already implemented an operation but used different variable names than those provided in the assignment. Finally, the feedback often centred around strictly following the approach from the model solution even when students maintained another (allowed and correct) method. For instance, an obvious solution for solving the brackets exercise is using a loop. One student was using join operations instead received several 'tips' to implement a loop. Or, for the pies exercises, we found students did not compute the total costs in cents, in contrast to the model solution. As a result, the students received:

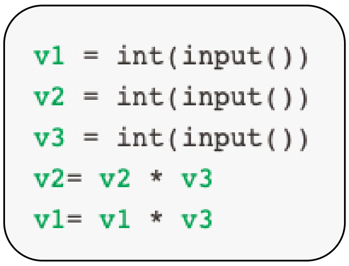
> "*The student's code seems to be attempting to calculate the cost of buying pies, but it is not following the same approach as the model solution. Try suggesting that the student should use the same formula as the model solution to calculate the total cost of buying pies.*"

We tried to indicate the solution as "Example solution" as we suspected "Model solution" might indicate that this is the only solution. However, this did not solve the problem.

The mentioned issues above occurred when a model solution was in the prompt, with or without a problem description. We, therefore, decided to continue our prompt design to use prompts with only a problem description. Despite not containing a solution, the model could still provide hints informing the students on their approach.

5.1.2 Instructions

While examining the effect of prompt attributes, we also analysed the influence of adding different instructions. We started with three different instruction phrasings, shown in table 5.1 with corresponding hints. Figure 5.1 shows the student program for which the hints were generated. Because we had no idea of effective prompts for hint generation, we first wanted an initial idea of how phrasing would affect the generated feedback. For this part, we did an explorative round where we documented our findings.



```
v1 = int(input())
v2 = int(input())
v3 = int(input())
v2= v2 * v3
v1= v1 * v3
```

Figure 5.1: Code snippet used to generate the hints from table 5.1

As illustrated in table 5.1, the prompt instructions generated output with the same subject but expressed differently. We did not discover when the instructions would produce feedback with other suggestions content-wise. In almost all cases, the generated outputs contained information on how to proceed, which was our starting point.

Accordingly, we analysed the results for three phrasings. We noticed specific patterns for each instruction. The prompt instructions had only slight variations among each other, consisting of a few different words. However, the generated feedback had notably different

Table 5.1: Example hints for different prompt instructions

Prompt	**Hint**
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.

results, especially regarding additional information and phrasing of the hints. The prompt: “What is the next step?” frequently contained code, more than the other two prompts. In addition, the phrasing was a bit more straightforward and blunt, which is also shown in table 5.1. Consequently, we considered this prompt less suitable for supporting students.

Contrarily, “Explain the next step for a student” resulted in relatively longer hints while sometimes also explaining the student’s code. However, even though code explanations can help students understand to trace code, we regarded the explanations as undesired. We aimed to assist with how to proceed, and the hint should focus on explaining that. 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 than the first instruction. Furthermore, this instruction produced feedback formulated as a hint without revealing the exact answer. For instance, “The next step involves computing the total costs in cents.” Based on all these findings, we decided to further the prompt engineering by combining keywords from the last two prompts, “hint” and “student”.

We again defined three prompts. Since we noted that the third prompt from the previous section had some good elements, such as its phrasing and references to the student code, we inserted the keyword “student” in our new instruction prompt. We decided to omit “explain” as we presume it caused too long feedback and undesirable code explanations. We decided to analyse the prompt: “Give this student a hint for the next step”. After briefly experimenting with this new prompt, we found it often generated relatively long outputs. So, we replaced it with two variations which instruct the model explicitly to create shorter hints. Since we initially found that changing only a few words to the prompt produced a notable difference in the phrasing, we included two new instructions instead of just one:

1. “Give a hint for the next step.”
2. “Give this student a short hint for the next step.”
3. “Give this student a hint for the next step. The hint should be one or two sentences.”

We used these three instructions for our experiments with the temperature parameter.

5.1.3 Temperature

First, we explored different temperature values for the “Give a hint for the next step” instruction (0.1, 0.3, 0.5, 0.7, and 0.9). We did a short comparison for prompts with only a description and a description combined with a model solution. We hoped to rule out that increasing the temperature would overcome the issues mentioned in section 5.1.1. However, with a brief analysis, we still encountered the same problems and thus chose to continue only with the description prompt.

To analyse the impact of temperature, we decided beforehand to focus on the effects per prompt and determined the optimal value for the temperature independently for each prompt. As the value for the temperature increased, we noted that hints tended to contain unusual or unexpected suggestions. For example, we saw several recommendations for using a stack in the brackets exercise, which is typically unnecessary and probably an unknown data structure to novel programmers. However, we saw improvements too, where hints were more tailored to the students and contained more references to their solutions. We counted 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.

5.2 Choosing the best prompt

To ultimately choose the best prompt, we ranked the three prompts from the previous section, all with a value for the temperature of 0.5. We did this in a team of two, this work’s author and the supervisor. We first randomly picked ten student programs for both exercises. Then, we compared the hints generated per program and ranked them from one to three, best to worst, respectively. The final score for each prompt was the sum of its total ratings, as shown in table 5.2. We decided to implement the most-right prompt from the table in our StAP-tutor.

Table 5.2: Ranking of prompts. We ranked the prompts with scores 1, 2, and 3, from best to worst, respectively.

	Give a hint for the next step	Give this student a short hint for the next step.	Give this student a hint for the next step. The hint should be one or two sentences.
Brackets	20	27	**16**
Pies	25	19	**17**
Total score	45	46	**33**

5.3 Best prompt practices

We discovered that LLMs might not always benefit from more information, as including a model solution led to feedback that forced specific approaches from the solution. Programming exercises have large solution spaces, and teachers can recognise valid solutions that differ from the model solution. Furthermore, they can support and help students without forcing them to change their approach. This adaptivity is probably not something that LLMs possess. Once provided with a model solution, LLMs create hints that are always related to that solution.

A similar error of LLMs was often to refer to the model solution as a hint, something a teacher would never do. Even when we specified in the prompt that it could not refer to the model solution, it still occurred several times. In their work, Hellas *et al.* encounter the same problem [18]. For now, we believe that LLMs cannot handle or interpret certain constraints very well. In another example, to shorten our hints, we included the sentence "use at most three sentences". This prompt produced only outputs with a length of exactly three sentences.

Regarding the phrasing, we found that using several keywords, such as "hint" and "student", can help with personal references and a friendly tone. Presentation is an important component of feedback. Students benefit from details and motivational words. Using these keywords, we found that the hints contained explanations, tips and sometimes 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.*".

Here, we see that the feedback compliments the student for correctly computing the total costs and provides extra tips for the operations the students can use. Additionally, using the keyword "hint" prevented hints that spoiled the answer, as we saw with using prompts "What is the next step?", which often produced code, while "Give a hint for the next step" gives tips such as "the next step involves using a for a loop."

However, one should be careful with formulating the instruction. One error we encountered was that the model could not recognise when the student completed the assignment and would always recommend a ‘next step’. We briefly experimented with adding something to the prompt to overcome this problem. For instance, we experiment with the prompt “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, we found that all generated feedback was much longer and had (too) many compliments. So adding “give a compliment” altered the complete outcome while we expected this to happen for only some of the hints. We, therefore, chose not to continue with this prompt in our experiments.

Finally, increasing the value for the temperature parameter might help produce better feedback. Yet, there is a 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 some tips are unnecessary and not helpful. Novice programmers might consider these hints the only truth and get dragged along with the proposed feedback. Nevertheless, increasing the temperature also resulted in more references to the student’s code and approach. Hence, choosing the temperature value depends on the context and should be carefully experimented with before applying.

We will conclude this chapter by reflecting on our formulated research question regarding prompt engineering:

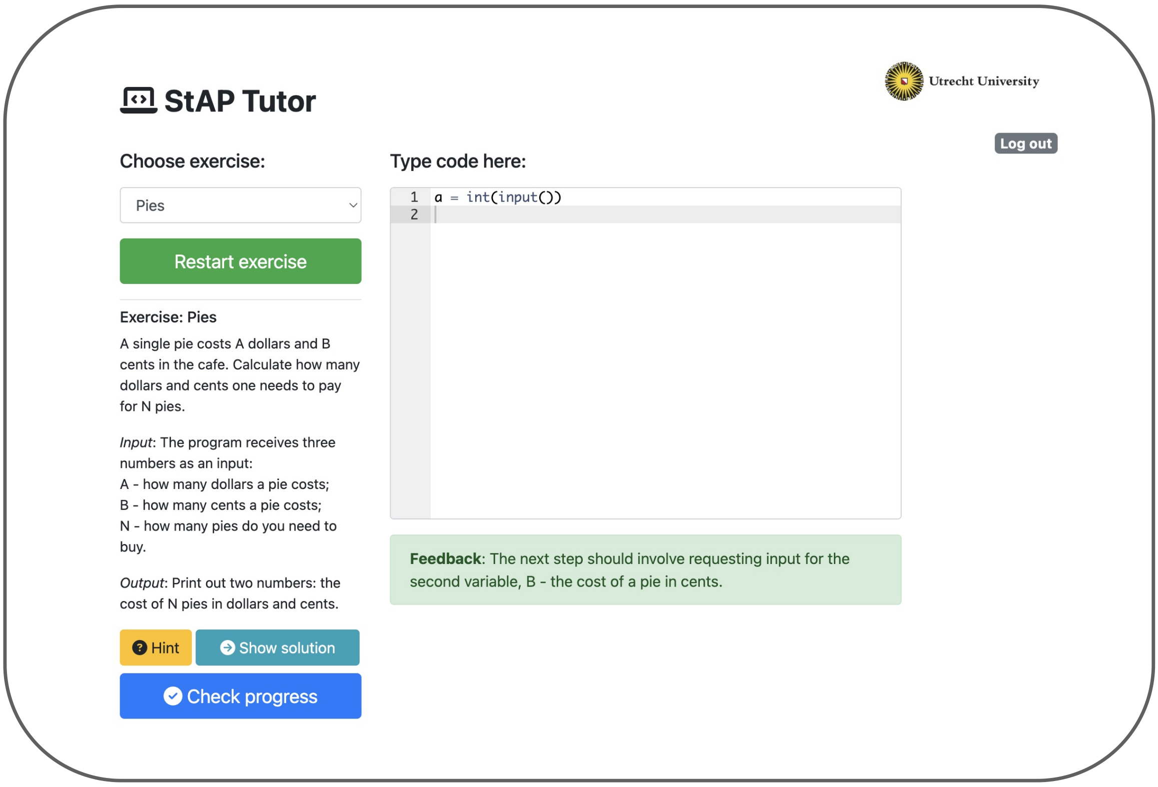
> **RQ2** What prompt characteristics are suitable for generating effective next-step hints with LLMs?

Prompt engineering is a complex process since there are limitless possibilities to experiment with, and the LLMs output will vary with every change. Our most meaningful findings are that including a problem description and keywords such as “hint” or “student” can help generate next-step hints. In addition, we found that increasing the value of the temperature parameter up until a certain level contributes to more personalized feedback. Furthermore, our work could serve as a foundation for prompt engineering practices to generate other types of automated feedback.

6 Evaluating next-step hints

6.1 StAP-tutor

After choosing and discovering the best prompt practices, we built and implemented the corresponding settings 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 Figure 6.1. 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, from which we chose the **gpt-3.5-turbo** model. Then, the API send the hint back to the interface, where it is displayed to the student.



Exercise: Pies

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 as an input:
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.

```
a = int(input())

```

Feedback: The next step should involve requesting input for the second variable, B - the cost of a pie in cents.

Figure 6.1: StAP Tutor interface

6.2 Student experiment

We first tested our tool with the student experiment. The students requested 11, 20 and 17 hints, which sum up to 48 in total. Examining the student's responses to the hint ratings, shown in Figure 6.2, we notice the students often marked the hints as clear and fitted to their work. However, the students were less convinced about the usefulness of the feedback. About only 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 by the student was for nothing.

Even though all students stated that the exercise corresponded with their current skills and programming level, not one student had a correct solution at the end of the experiment. The students had especially trouble with the logic for correctly counting the clumps, which required tracking and implementing when to 'count' a clump with conditional statements.

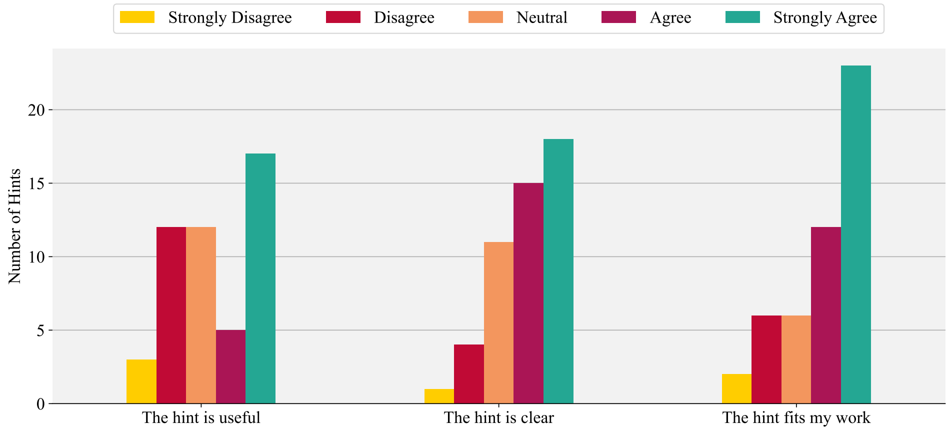


Figure 6.2: Student ratings per question

6.2.1 Overall students' impression

After the experiment, we asked the students several final questions about their experiences and opinions. Overall, they had a positive impression of the StAP-tutor and appreciated the hints, especially at the beginning of the experiment. The students found that at the start of the assignment, the feedback fits the student's 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 stated they needed more detail or practical suggestions, while the feedback remained too vague. In addition, the students indicated they found it difficult to program without additional information a standard IDE provides, such as simple syntax errors or the output generated by their code.

Notably, all students mentioned using the StAP-tutor to request multiple hints for the same code. As the language model is not deterministic, it might help to regenerate a hint when it is too vague or not necessarily helpful. They reported that regenerating creates a different phrasing or provides more or new information. However, one of the students stated that requesting the same 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 much additional information. The students thought this might help with learning as this also reduces the temptation to copy and paste the assignment and generate a whole solution. However, as one of the students points out, an advantage of ChatGPT is that there is a possibility to ask more specific questions yourself. The StAP-tutor automatically generates the prompts and has no opportunity for student input.

6.3 Expert assessment

We started our expert evaluation by analysing if we had similar interpretations of the evaluation criteria. We randomly selected a group of 19 hints and compared our ratings based on the criteria, for which the agreements are shown in table 6.1. Additionally, we computed the corresponding inter-rater reliability with Cohen's Kappa. According to a proposed interpretation from Landis *et al.*, we interpret values from 0.21-0.40 as fair, 0.41-0.60 as moderate, 0.61-0.80 substantial and 0.81-1.00 as almost perfect [24]. Overall, we consider that there exists a substantial agreement between the ratings of the two experts since we disagreed on only 13 out of the 152 values. The criteria *Level-of-detail* and *Tone* had the lowest score, which was expected as these are assumably relatively subjective categories. We discussed and made agreements on the clashing values. Then, we merged the ratings with those for the remaining 29 hints. These hints were rated by only the author of this work. We will analyse these results in the remainder of this section.

Table 6.1: Inter-rater reliability for the expert evaluation

	Feedback Type	**Information**	**Level-of-detail**	**Personalized**
Agreements	19/19	18/19	16/19	18/19
Cohen's kappa	0.872	0.787	0.387	0.883

	Appropriate	**Specific**	**Misleading Information**	**Tone**
Agreements	18/19	18/19	17/19	15/19
Cohen's kappa	0.855	0.642	0.683	0.467

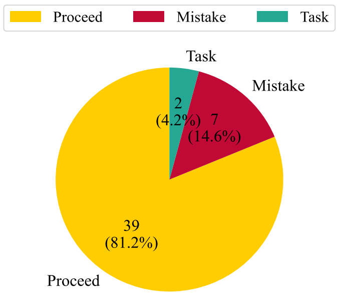


Figure 6.3: Feedback Type

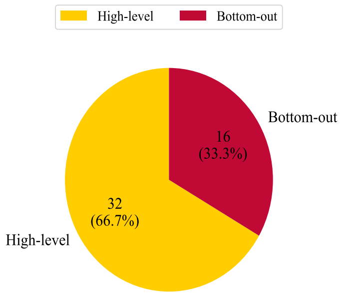


Figure 6.4: Level of detail

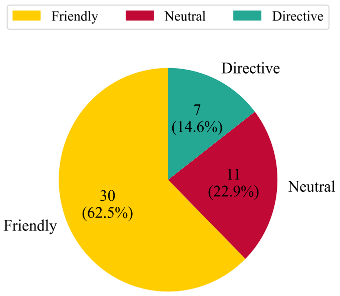


Figure 6.5: Tone

Looking at figures 6.3, 6.4, and 6.5, we see 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*, which is ultimately the goal of this research project. Moreover, we found that the hints are more often high-level than bottom-out. Note, in contrast with the type of feedback, we did not particularly specify in the prompt the level of detail we desired. Still, the model tended to generate feedback with more high-level hints with these specific settings. Finally, we found that the feedback mostly had a friendly tone. We interpreted the tone as friendly when the hints were more suggestive than directive. For example: "You consider trying to change the condition of the for loop" instead of "Change the for loop condition".

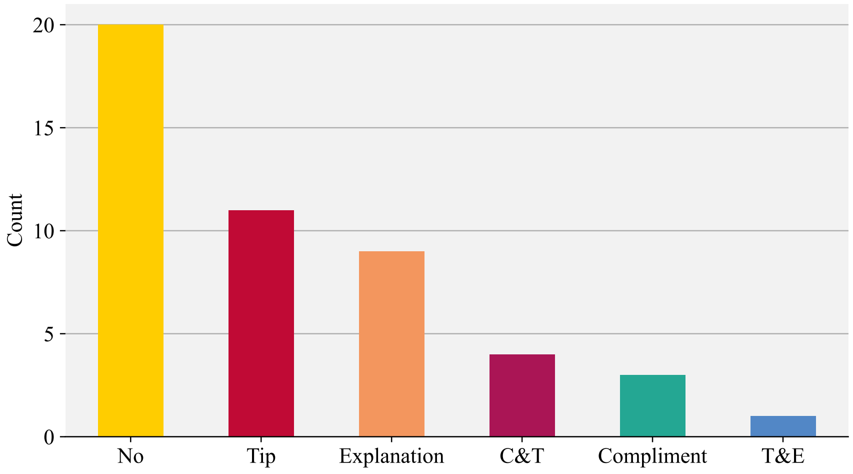


Figure 6.6: Counts for the additional information criteria. Categories C&T and T&E are a combination of additions. So, a compliment and tip, or tip with explanation, respectively.

We found that friendly hints often contain additional information. Naturally, when feedback has a motivating statement such as "Keep up the good work!", we interpret this as friendly or kind. Figure 6.6 shows the frequencies for every additional information category. We occasionally noticed that feedback contained smaller tips in addition to the 'main' hint. These tips would often refer 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 6.7. We consider these positive results, as feedback is frequently personalised, appropriate and specific. We saw hints with explicit references to variables from the student code and suggestions that referred to the current implementation of the student. 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 more often subtle. For example, feedback explained incorrectly why some part 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.

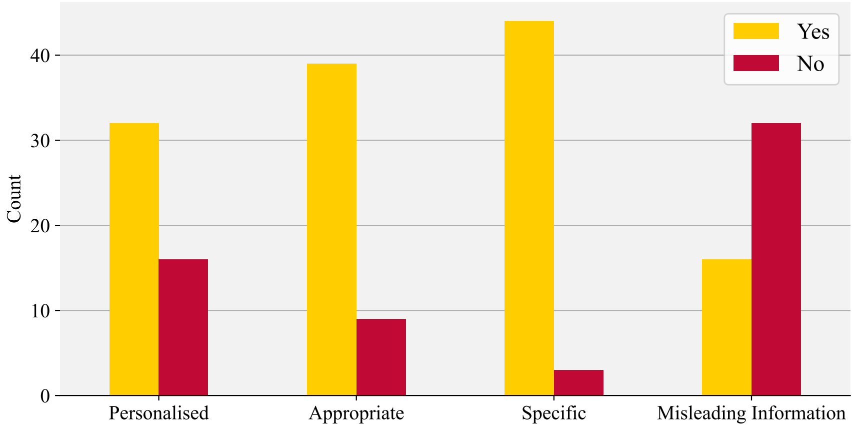


Figure 6.7: Frequencies of binary evaluation metrics: Personalised, Appropriate, Specific and Misleading Information

6.4 The quality of next-step hints generated by LLMs

We found overall that the outcomes have promising results. A substantial part of the feedback was specific, which we see as a positive result since too complex feedback can prevent students from attentively reading and comprehending the provided information. Additionally, for solving programming exercises, one can break down the problem into smaller sub-tasks. This strategy helps students realise how to break down a complex problem into subtasks and perform a sequence of operations that lead to a solution [23]. Students agreed the hints were overall clear, and most feedback consisted of 1 or 2 sentences, with some outliers to three sentences. Hence, we believe that LLM-generated feedback conforms to the right level of complexity and limited information and is easy for the students to understand. We showed that LLMs can respond to students' help requests with appropriate and potentially helpful answers, similar to the work of Hellas *et al.* [18]. Furthermore, the hints are overall easy to understand, as LLM-generated explanations [27]. This work joins other research by highlighting the potential and utility of applying LLMs in programming education.

It was interesting to see how students interacted with the StAP-tutor. They let the tool create multiple hints for the same code without intermediate changes. According to the students, this would help when feedback was not clear or helpful. An advantage of using LLMs for hint generation, their responses are not deterministic as opposed to other data-driven hints [49, 52]. Hence, they can create various distinctive hints with the same prompt. Depending on the prompt and the temperature, these hints may vary both in phrasing and content. A different formulation might help the student understand the feedback if it was initially not clear.

However, all students indicated they needed several times more support than provided during the experiment. The used prompt tends to generate high-level hints. These hints can help the students get started and support them with constructing an idea for an overall approach. Yet, when approaching the end of the exercise, it is only logical that students who are stuck do not benefit from feedback that explains how to. And in these cases, it might be necessary to use a different prompt instead of generating multiple hints with the same prompt. We will elaborate on this in section 7.3. Contrarily, another demonstrated advantage of our prompt practices is they frequently include additional information, such as compliments, tips and explanations. Furthermore, students mentioned they liked how the hints occasionally added some 'motivational words', which made the interaction feel more personal.

An obvious limitation of the current performance of LLM-generated hints is that they occasionally contain misleading information. This issue is hard to resolve with further prompt engineering as we believe it is not directly correlated to prompt practices but relates to the overall capability of the LLM itself. We expect that as the performance of LLM increases, what most certainly will happen, these limitations will resolve on their own. However, when we use LLM in the context of teaching, we should be cautious about them spreading incorrect information, as this could negatively impact the student's learning. We found it important the feedback was not overly confident and directive, so the student was encouraged to think critically about the hint themselves.

> **RQ3** What are students' and experts' perceptions of the quality of LLM-generated next-step hints, and how do they rate them?

Both students and experts have positive impressions about the potential of using LLMs for generating next-step hints. While the feedback succeeded at providing tailored support well-fitted to the student, it was not always helpful and may contain misleading information.

7 Discussion

So far, we have discussed the method and results of our work. We made suggestions for the best prompt practices and reflected on the quality of the generated hints from the StAP-tutor. This chapter first reviews our main research question. Then, we will discuss the limitations of our research regarding our prompt-engineering process, experimental set-up and reproducibility of this work. Hereafter, we address how future research can address these limitations and discuss several other directions for future work.

7.1 Generating next-step hints using LLMs

This work ultimately aims to put perspective on using LLMs in programming education. More specifically, contributing work exploring other applications, we examined possibilities concerning generating next-step hints.

> **RQ1** To what extent can we use LLMs to generate informative and effective next-step hints for Python introductory exercises?

We found that the best method for generating next-step hints is by using prompts that refer to a problem description and include keywords related to the context of this application, such as “student” or “hint”. Furthermore, we can use LLM to generate personalised feedback referring to the student’s code and approach. Compared to other data-driven approaches, these models can generate personalised feedback with additional information while they do not require large datasets or complex computational methods. In contrast with standard hints, which 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 the LLM-generated feedback now may entail misleading information. We expect that as the performance of LLM improves, this issue will be less occurring. However, we should be cautious about using these incorporating automatically generated feedback as it might harm the students’ understanding.

7.2 Threats to validity

7.2.1 Prompt engineering

As mentioned earlier, most of the studies that used LLMs in the context of programming education did not report anything regarding prompt engineering and their evaluation.

Although we aimed for an extensive engineering process, we had to make some limiting decisions. 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 prompt characteristics independently. However, we acknowledge that these factors can influence each other. For example, we first found that including a model solution was not beneficial. However, this decision was based on a low temperature. We did a short exploration to see whether the same problems would hold for higher temperature values, but we did not extensively research this. As we did not observe clear improvements, we ultimately decided not to include model solutions. Nevertheless, this example illustrates our approach is not exhaustive, but we aimed to provide an overview of possible effective prompt practices for generating programming feedback.

Secondly, we have only analysed a small set of prompt instructions by experimenting with a few keywords. As we were content with the preliminary results, we chose not to explore more instructions with completely different phrasings. Using other formulations could have led to hints of different qualities. We chose an approach with relatively minor changes as we registered that these would still impact the output of the LLM. With this method, we aimed to provide guidelines for choosing a prompt instead of analysing the behaviour of a large set of prompts.

Lastly, we executed the first part of prompt engineering without strict evaluation. We created hints for two exercises, multiple students and code snapshots. We aimed for a general impression of how the LLM would perform for a wide variety of code states. Consequently, we did not generate multiple hints with the same settings for the student programs as this was not feasible to do as well within our given time frame. As LLM are non-deterministic, the output may variate for the same prompt, which we did not analyse. However, as we had consistent findings in our evaluation, we believe that despite our method being more explorative, our proposed guidelines for prompt engineering on this topic are valid.

7.2.2 Experimental setup

Regarding the evaluation, we performed the student experiments with just three students due to a lack of time and voluntary participation. Furthermore, we only examined the students' opinions instead of objective factors indicating improved learning. We, therefore, can not make strong statements about the effect of the hints. Such conclusions require a long-term study on a large group of students and examination of pre-and post-tests with a control group. However, setting up the evaluation of intelligent tutoring systems is a complex task [2]. Additionally, analysis of students' perceptions is currently the most common method for evaluating the hints' effectiveness [8]. Future work could support this research with an extended evaluation.

In addition to the student experiments, we performed expert assessments. We aimed to capture as many important factors for effective feedback into our evaluation criteria, supported by the literature from section 2.1. 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, while not necessarily proven to be the best practice [20]. Implementing other methods for timing the feedback, however, require a method for automating hint delivery, which was outside the scope of this work. We discuss this and other possible extensions in section 7.3.

7.2.3 Reproducibility

Several studies on LLM-related topics face the problem that these models are constantly updated, which might produce other results. During the execution of our experiments, OpenAI released GPT-4. It was not feasible anymore to implement this new model in our work. However, we did perform some minor experiments using GPT-4 and compared its results with those generated by the GPT-3.5-turbo API. We found that the same prompt for both models outputs, at first glance, different results. Different results do not necessarily imply that the prompts found effective during this work will not work for another model. It is unknown how prompts translate to new models. For instance, the phrasing and content may vary and give other results than was desired. However, we primarily found that 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 hypothesis.

7.3 Future work

We believe this work sets out various possibilities for future work regarding generating (next-step) hints in programming education. Although not perfect, at this time, LLM-generated hints certainly have potential, and we can explore plenty of options in further research. In this section, we mention merely a few suggestions, yet there are many directions to continue from this work.

As the students pointed out during their experiment, in some cases, the hints were not as helpful as they needed details. Students may benefit from help described at different levels of information depending on the context. Once students are stuck, they might require more in-depth feedback that offers better guidance than high-level hints. During prompt engineering, we did not aim to generate hints with a specific level of detail beforehand, as we had no prior knowledge of what the prompts would generate. Our StAP-tutor eventually used only one prompt for hint generation. The results demonstrated that our chosen prompt outputs both bottom-out and high-level hints.

With this knowledge, future work could explore prompt engineering further, investigating the relation between prompts and the level of detail from the hints they generate. On another note, one could study generating feedback according to the classification provided by Keuning *et al.* [22], as mentioned in 2.3. To take this even a step further, one could combine these findings with advanced methods for student modelling to provide personalised feedback and guidance based on the student's progress. For instance, depending on the number of hints requested for a specific state, a digital learning environment could use a different prompt to obtain feedback providing more guided assistance. Other possible advances are to generate positive and other types of feedback. Or to create hints for more intermediate programming exercises. Implementing these ideas was out of the scope of this research.

Another missing feature pointed out by students was the ability to influence the topic of hints, which is possible when a student uses ChatGPT directly. Prather *et al.* found similar frustrations by students while they work with Copilot [46]. We also noticed during evaluation that hints occasionally did not correspond to the part of the code that students were struggling with at that time. For example, the StAP-tutor suggested writing already-written pieces of code. As a next step, the StAP-tutor could implement student control. For instance, the interface could include an option for the students to indicate what exact part of the code they want help with. Then, in further research, one could examine how to present this additional information in the prompt such that the hints address that specific part of the code.

More subtle suggestions for future work are adding few-shot learning or experimenting with providing multiple solutions in the prompts. Both approaches were beyond the scope of this research. To improve the performance of LLMs, one can include few-shot-learning during the prompting process [4]. With few-shot-learning, one includes examples, so-called shots, in the prompts. LLM could then learn and follow the structure or pattern provided in the shots. However, we required a set of expert-validated example hints to implement this approach, which we did not have for our dataset. Such annotated datasets are rare but certainly have value for programming education research. Letting LLMs learn from expert feedback might improve the quality of LLM-generated hints even more.

8 Conclusion

In this work, we investigated how to use LLMs to generate next-step hints for introductory programming exercises in Python. 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 value for the temperature parameter. According to our findings, 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 for the temperature may cause odd and unhelpful suggestions.

We then created the StAP-tutor based on these results. With our tool, students can practice programming with multiple exercises. The ratings and discussion from our student experiment indicated that students 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.

Although we employed two evaluation methods, we need a more methodical examination of the hints’ quality. We can only then make validated statements about the effectiveness of the LLM-generated feedback. Another limitation of this work is that the feedback was generally high-level, which becomes less valuable as students proceed with solving the problem. One suggestion for future work is to examine whether other prompts may be more effective in these cases and eventually recognise such situations. Furthermore, we can expand our work by investigating generating multiple feedback types, incorporating student control in the prompts or experimenting with few-shot learning.

In conclusion, our study highlights the potential of using LLM in programming education by generating next-step hints. In conjunction with other related research, we demonstrated that LLMs can have various applications within programming education by supporting both students and teachers. Therefore, we should not concentrate on the complications of these advances but on the practical value they bring to education in general instead.

Bibliography

[1] Tiffany Barnes and John Stamper. “Toward automatic hint generation for logic proof tutoring using historical student data”. In: *Intelligent Tutoring Systems: 9th International Conference, ITS 2008, Montreal, Canada, June 23-27, 2008 Proceedings 9*. Springer. 2008, pp. 373–382.

[2] Joseph E Beck et al. “Does help help? Introducing the Bayesian Evaluation and Assessment methodology”. In: *Intelligent Tutoring Systems: 9th International Conference, ITS 2008, Montreal, Canada, June 23-27, 2008 Proceedings 9*. Springer. 2008, pp. 383–394.

[3] Brett A Becker et al. “Programming Is Hard-Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation”. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 500–506.

[4] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[5] Mark Chen et al. “Evaluating Large Language Models Trained on Code”. In: *arXiv e-prints* (2021), arXiv–2107.

[6] Jacob Cohen. “A coefficient of agreement for nominal scales”. In: *Educational and psychological measurement* 20.1 (1960), pp. 37–46.

[7] Phillip Dawson et al. “What makes for effective feedback: Staff and student perspectives”. In: *Assessment & Evaluation in Higher Education* 44.1 (2019), pp. 25–36.

[8] Galina Deeva et al. “A review of automated feedback systems for learners: Classification framework, challenges and opportunities”. In: *Computers & Education* 162 (2021), p. 104094.

[9] Paul Denny, Viraj Kumar, and Nasser Giacaman. “Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language”. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 1136–1142.

[10] Paul Denny et al. “Computing Education in the Era of Generative AI”. In: *arXiv preprint arXiv:2306.02608* (2023).

[11] Jean-Baptiste Döderlein et al. “Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic?” In: *arXiv e-prints* (2022), arXiv–2210.

[12] Zhangyin Feng et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. 2020, pp. 1536–1547.

[13] James Finnie-Ansley et al. “My AI Wants to Know if This Will Be on the Exam: Testing OpenAI’s Codex on CS2 Programming Exercises”. In: *Proceedings of the 25th Australasian Computing Education Conference*. 2023, pp. 97–104.

[14] James Finnie-Ansley et al. “The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming”. In: *Australasian Computing Education Conference*. 2022, pp. 10–19.

[15] Davide Fossati et al. “Data driven automatic feedback generation in the iList intelligent tutoring system”. In: *Technology, Instruction, Cognition and Learning* 10.1 (2015), pp. 5–26.

[16] Qiang Hao et al. “Towards understanding the effective design of automated formative feedback for programming assignments”. In: *Computer Science Education* 32.1 (2022), pp. 105–127.

[17] John Hattie and Helen Timperley. “The power of feedback”. In: *Review of educational research* 77.1 (2007), pp. 81–112.

[18] Arto Hellas et al. “Exploring the Responses of Large Language Models to Beginner Programmers’ Help Requests”. In: *arXiv preprint arXiv:2306.05715* (2023).

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

[20] Johan Jeuring et al. “Towards Giving Timely Formative Feedback and Hints to Novice Programmers”. In: *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education*. 2022, pp. 95–115.

[21] Majeed Kazemitabaar et al. “Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–23.

[22] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. “A systematic literature review of automated feedback generation for programming exercises”. In: *ACM Transactions on Computing Education (TOCE)* 19.1 (2018), pp. 1–43.

[23] Theodora Koulouri, Stanislao Lauria, and Robert D Macredie. “Teaching introductory programming: A quantitative evaluation of different approaches”. In: *ACM Transactions on Computing Education (TOCE)* 14.4 (2014), pp. 1–28.

[24] J Richard Landis and Gary G Koch. “The measurement of observer agreement for categorical data”. In: *biometrics* (1977), pp. 159–174.

[25] Timotej Lazar and Ivan Bratko. “Data-driven program synthesis for hint generation in programming tutors”. In: *Intelligent Tutoring Systems: 12th International Conference, ITS 2014, Honolulu, HI, USA, June 5-9, 2014. Proceedings 12*. Springer. 2014, pp. 306–311.

[26] Nguyen-Thinh Le. “A classification of adaptive feedback in educational systems for programming”. In: *Systems* 4.2 (2016), p. 22.

[27] Juho Leinonen et al. “Comparing Code Explanations Created by Students and Large Language Models”. In: *arXiv preprint arXiv:2304.03938* (2023).

[28] Juho Leinonen et al. “Using large language models to enhance programming error messages”. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 563–569.

[29] Yujia Li et al. “Competition-level code generation with alphacode”. In: *Science* 378.6624 (2022), pp. 1092–1097.

[30] Raymond Lister, Colin Fidge, and Donna Teague. “Further evidence of a relationship between explaining, tracing and writing skills in introductory programming”. In: *Acm sigcse bulletin* 41.3 (2009), pp. 161–165.

[31] Elena Lyulina et al. “TaskTracker-Tool: A Toolkit for Tracking of Code Snapshots and Activity Data During Solution of Programming Tasks”. In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. SIGCSE ’21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 495–501. ISBN: 9781450380621. DOI: `10.1145/3408877.3432534`. URL: `https://doi.org/10.1145/3408877.3432534`.

[32] Stephen MacNeil et al. “Experiences from using code explanations generated by large language models in a web software development e-book”. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 931–937.

[33] Stephen MacNeil et al. “Generating diverse code explanations using the gpt-3 large language model”. In: *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2*. 2022, pp. 37–39.

[34] Yana Malysheva and Caitlin Kelleher. “An Algorithm for Generating Explainable Corrections to Student Code”. In: *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*. 2022, pp. 1–11.

[35] Samiha Marwan et al. “The impact of adding textual explanations to next-step hints in a novice programming environment”. In: *Proceedings of the 2019 ACM conference on innovation and technology in computer science education*. 2019, pp. 520–526.

[36] Antonija Mitrovic, Kenneth R Koedinger, and Brent Martin. “A comparative analysis of cognitive tutoring and constraint-based modeling”. In: *User Modeling 2003: 9th International Conference, UM 2003 Johnstown, PA, USA, June 22–26, 2003 Proceedings 9*. Springer. 2003, pp. 313–322.

[37] Elham Mousavinasab et al. “Intelligent tutoring systems: a systematic review of characteristics, applications, and evaluation methods”. In: *Interactive Learning Environments* 29.1 (2021), pp. 142–163.

[38] Susanne Narciss. “Feedback strategies for interactive learning tasks”. In: *Handbook of research on educational communications and technology*. Routledge, 2008, pp. 125–143.

[39] Florian Obermüller, Ute Heuer, and Gordon Fraser. “Guiding next-step hint generation using automated tests”. In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. 2021, pp. 220–226.

[40] OpenAI. “GPT-4 Technical Report”. In: *ArXiv* abs/2303.08774 (2023).

[41] OpenAI. *OpenAI Codex*. URL: `https://openai.com/blog/openai-codex/`. (accessed: 30.12.2022).

[42] Benjamin Paassen et al. “The Continuous Hint Factory-Providing Hints in Vast and Sparsely Populated Edit Distance Spaces”. In: *Journal of Educational Data Mining* 10.1 (2018), pp. 1–35.

[43] Nick Parlante. *CodingBat*. URL: `https://codingbat.com/python`. (accessed: 24.02.2023).

[44] Neil Perry et al. “Do Users Write More Insecure Code with AI Assistants?” In: *arXiv preprint arXiv:2211.03622* (2022).

[45] Tung Phung et al. “Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models”. In: *arXiv preprint arXiv:2302.04662* (2023).

[46] James Prather et al. “” It’s Weird That it Knows What I Want”: Usability and Interactions with Copilot for Novice Programmers”. In: *arXiv preprint arXiv:2304.02491* (2023).

[47] Julian Aron Prenner and Romain Robbes. “Automatic Program Repair with OpenAI’s Codex: Evaluating QuixBugs”. In: *arXiv preprint arXiv:2111.03922* (2021).

[48] Thomas W Price, Yihuan Dong, and Tiffany Barnes. “Generating data-driven hints for open-ended programming.” In: *International Educational Data Mining Society* (2016).

[49] Thomas W Price, Yihuan Dong, and Dragan Lipovac. “iSnap: towards intelligent tutoring in novice programming environments”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on computer science education*. 2017, pp. 483–488.

[50] Thomas W Price et al. "A comparison of the quality of data-driven programming hint generation algorithms". In: *International Journal of Artificial Intelligence in Education* 29 (2019), pp. 368–395.

[51] Thomas W Price et al. "Factors influencing students' help-seeking behavior while programming with human and computer tutors". In: *Proceedings of the 2017 ACM Conference on international computing education research.* 2017, pp. 127–135.

[52] Kelly Rivers and Kenneth R Koedinger. "Data-driven hint generation in vast solution spaces: a self-improving python programming tutor". In: *International Journal of Artificial Intelligence in Education* 27 (2017), pp. 37–64.

[53] Sami Sarsa et al. "Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models". In: *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1.* 2022, pp. 27–43.

[54] Valerie J Shute. "Focus on formative feedback". In: *Review of educational research* 78.1 (2008), pp. 153–189.

[55] Michael Striewe and Michael Goedicke. "A review of static analysis approaches for programming exercises". In: *Computer Assisted Assessment. Research into E-Assessment: International Conference, CAA 2014, Zeist, The Netherlands, June 30–July 1, 2014. Proceedings.* Springer. 2014, pp. 100–113.

[56] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models". In: *Chi conference on human factors in computing systems extended abstracts.* 2022, pp. 1–7.

[57] Benedikt Wisniewski, Klaus Zierer, and John Hattie. "The power of feedback revisited: A meta-analysis of educational feedback research". In: *Frontiers in Psychology* 10 (2020), p. 3087.

[58] Junaed Younus Khan and Gias Uddin. "Automatic Code Documentation Generation Using GPT-3". In: *arXiv e-prints* (2022), arXiv–2209.

[59] Jialu Zhang et al. "Repairing Bugs in Python Assignments Using Large Language Models". In: *arXiv preprint arXiv:2209.14876* (2022).