Exploring How Multiple Levels of GPT-Generated Programming Hints Support or Disappoint Novices

Ruiwei Xiao
ruiweix@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Xinying Hou
xyhou@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

John Stamper
jstamper@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

ABSTRACT

Recent studies have integrated large language models (LLMs) into diverse educational contexts, including providing adaptive programming hints, a type of feedback focuses on helping students move forward during problem-solving. However, most existing LLM-based hint systems are limited to one single hint type. To investigate whether and how different levels of hints can support students' problem-solving and learning, we conducted a think-aloud study with 12 novices using the LLM Hint Factory, a system providing four levels of hints from general natural language guidance to concrete code assistance, varying in format and granularity. We discovered that high-level natural language hints alone can be helpless or even misleading, especially when addressing next-step or syntax-related help requests. Adding lower-level hints, like code examples with in-line comments, can better support students. The findings open up future work on customizing help responses from content, format, and granularity levels to accurately identify and meet students' learning needs.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in HCI**.

KEYWORDS

Programming Hint, Large Language Model, GPT, Introductory Programming, Help-seeking

ACM Reference Format:
Ruiwei Xiao, Xinying Hou, and John Stamper. 2024. Exploring How Multiple Levels of GPT-Generated Programming Hints Support or Disappoint Novices. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '24), May 11–16, 2024, Honolulu, HI, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3613905.3650937

1 INTRODUCTION

Programming novices need in-time, effective support when getting stuck. Intelligent programming tutors (IPTs) can generate adaptive feedback, which can be a scalable solution to beginners' high demands for help. In the past two years, LLMs have been demonstrated to be effective in generating code fixes and explanations, leading to their application in programming tutors for generating hints to students [15]. Current LLM-based systems mainly focus on leveraging the language model's advantage to provide hints as formative feedback in more descriptive, high-level natural language, imitating the human teaching assistant's tone [13].

Despite the benefits of providing multiple levels of hints in intelligent tutoring systems studies [1], few studies investigated the diversity in LLM-generated programming hints or how to deliver these hints to maximize student learning. To bridge these gaps, we leveraged existing literature on how to deliver formative feedback [26, 28] to develop the LLM Hint Factory, which provides 4 different levels of LLM-generated hints in a thread for each hint request, from general natural language guidance to concrete code snippet suggestion. We then conducted a user study with 12 novice programming learners to complete three programming learning tasks with pre- and post-test using the LLM Hint Factory. Results showed that, delivering high-level natural language hints alone can be insufficient, or sometimes even harmful to students, especially when students are confused about next-step logic or syntax-related details. Adding low-level hints, particularly example code pieces with in-line comments can optimally help students in most cases. This work provides a novel contribution by designing the LLM Hint Factory, a system to provide scalable, high-quality programming hints at various levels, and revealing patterns of appropriate hint levels for varied help-seeking circumstances and the corresponding reasons. This work also contributes to the field by highlighting the need to distinguish students' demands based on their help-seeking contexts. The response to students' help requests should be personalized in terms of content, format, and granularity to meet students' diverse needs. It also opens up work into what and how future instructional agents, including educational chatbots and AI teaching assistants, should be designed to respond to students' diverse help requests optimally.

2 RELATED WORK

2.1 Hint delivery in traditional intelligent programming tutors

Hint is a special type of formative feedback. Hints provided in programming problem-solving mostly focus on providing information to guide learners towards the next steps for a correct solution [19]. In traditional IPT research, different levels of hints have been implemented with theoretical support and examined by empirical evidence. Hints in some systems are designed in high-level natural language with limited programming syntax to simulate human tutors. For instance, hints from a rule-based IPT Lisp tutor are provided in human tutors' tone to explain how to do next [2]. Another system, ITAP, is a data-driven Python programming tutor that also provides one-sentence next-step suggestions in natural language with location, value, action, and context information [22].

In contrast to providing natural language hints, other tutoring systems provide more comprehensive and concrete support, such as worked example code (a code solution to a similar problem) and bottom-out code (the code solution to this problem) hints to reduce students' cognitive load. For example, iSnap [21] presents the student's buggy part of the code with suggested next-step code side by side to demonstrate what to do next. In addition, Yahoo! Pipes [14] provides external links containing worked examples to learners as hints. What's more, programming tutors that provide on-demand Parsons problems related to the current question can be content-wise seen as button-out code hints when students seek for certain syntax on one of the lines or finish solving the Parsons problem [7, 8]. Moving from single hint type to multi-levels of hints, Suzuki et al. identified 5 types of hints through program synthesis [29]. Providing multiple levels of hints for one request not only provides learners with extended sources of help, but also allow them to compare the helpfulness of different levels of hints under different conditions. For example, Hint Factory [27] derives a hint sequence into 4 parts: three of them focus on one of the certain components in the question structure and one bottom-out hint on the explicit action needed to proceed to the next step. What's more, Ask-Elle [6] organized descriptive and worked-out code hints of the next step for all potential problem-solving strategies.

Regardless of the varied hint designs and their promising results to support student learning, scalability remains a key challenge across all levels of hint generation in traditional IPT systems. Conventionally, delivering adaptive hints either require taxing expert inputs (e.g. annotating answers [6], defining rules [2], etc.), or need to be driven by a high volume, comprehensive dataset [22, 29]. The emergence of LLMs provides a new opportunity to automate the hint generation process and customize the hints to target students' diverse needs.

2.2 LLM-Based Intelligent Programming Tutors

After recognizing the ability of LLMs to generate code fixes [17, 25], explanations [3], and pedagogical conversations [30], significant effort has been invested in the creation and evaluation of LLM-based intelligent programming tutors with diverse content focus and granularity. Most existing studies are dedicated to providing higher-level hints in paragraphs of natural language and cautiously avoid providing concrete solution-like code to prevent overuse or over-reliance. However, some of these studies acknowledged that students prefer to have hints with varied levels of details than high-level ones alone [15, 23]. Meanwhile, Kazemitabaar et al. [9] pointed out that having access to the AI code generator and generated code snippets did not impede learning gains. Therefore, we believe that code examples for similar problems have the potential to be incorporated as a more specific level of hints. Among the recent deployment of LLM-based IPTs in intro CS education, to our knowledge, no work has explored the effects of different hint levels on student problem-solving. We identified only one study that provided three levels of code explanation (line-by-line, summary, and key concepts) to students side-by-side in an E-book [17]. However, explanations in this system are additional instructional explanations for instructional materials rather than hint feedback to guide students to move forward in problem solving. Our work distinguishes itself from existing approaches by comparing the effects between different levels of next-step hints on supporting novices' problem-solving. More specifically, our system (the LLM Hint Factory) provides four levels of hints for each hint request, giving students full autonomy to use any hint level and ask for hints whenever needed. Hint level design follows principles and definitions in previous formative feedback work with an emphasis on programming exercises[11].

3 LLM HINT FACTORY

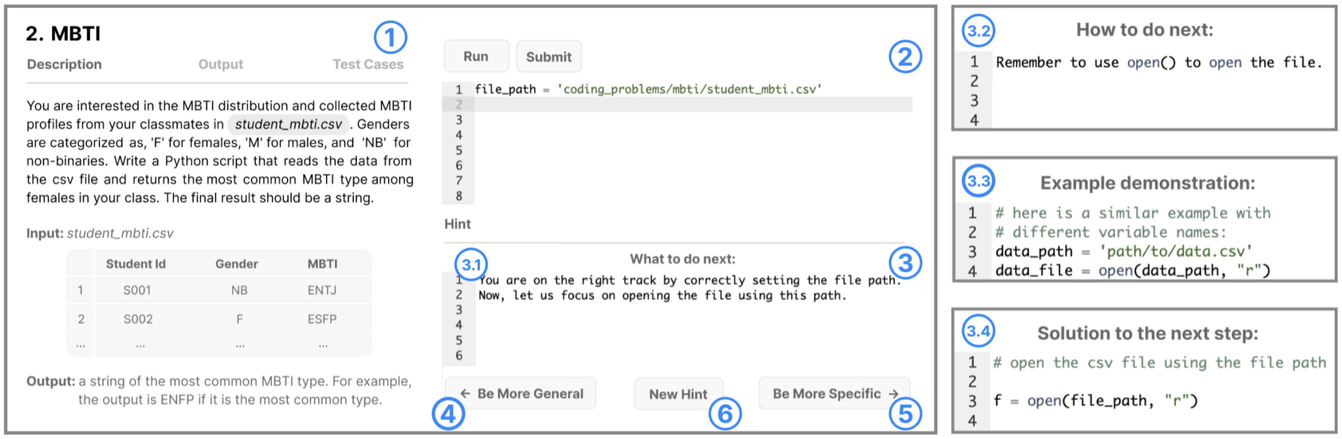
Based on our goals, we designed and developed the LLM Hint Factory, a system that provides four levels of hints, from the most high-level natural language guidance to the most specific code snippets, to CS1 students in real-time when they ask for help in solving a programming task (Figure 1).

Hint Design in the LLM Hint Factory Programming hints in the LLM Hint Factory are formative, next-step feedback within an economy of words [5]. Compared to many existing LLM-based IPTs, which provide long explanations mixed with small snippets of code trying to resolve all issues in the student's code, our feedback is segmented into individual steps to break the information into manageable chunks. By doing this, we aim to help learners manage their essential processing in working memory to achieve better learning. Given that even the bottom-out hints only contain part of the solution, this can also prevent the potential over-reliance by showing students a full LLM-generated answer [10]. Specifically, we designed four levels of hints, from the highest level to the most specific level: orientation hint, instrumental hint, worked example hint, and bottom-out hint. The division of levels is originally inspired by the hint-level design and definition in Cognitive Tutor [12], and we further grounded the definition into programming context and added one more level, worked example hint into the LLM Hint Factory. Prompts for different levels' hints generation can be found in Appendix A.

LLM Hint Factory Interface LLM Hint Factory shares great similarities to a typical online programming environment (Figure 1). The left panel includes the problem description and a clickable link to check the input data; the right panel contains a code editor and a "Hint" tab. Students can click "New Hint" to generate a set of four hints based on the current question information and code state, and click "Be More Specific" to read a more explicit hint targeting the same sub-step or click "Be More General" to navigate to the last hint level. The button-triggering for more specific level of hint design adheres to scaffolding guidelines in learning sciences research [18], which suggest to provide just right level of assistance needed by each individual.

4 METHOD

To understand the effectiveness of diverse levels of programming hints, we conducted an IRB (Institutional Review Board)-approved think-aloud study with a pre-test and post-test to investigate how these hints could help students with programming problem-solving and contribute to learning.



2. MBTI

You are interested in the MBTI distribution and collected MBTI profiles from your classmates in *student_mbti.csv*. Genders are categorized as, 'F' for females, 'M' for males, and 'NB' for non-binaries. Write a Python script that reads the data from the csv file and returns the most common MBTI type among females in your class. The final result should be a string.

Input: *student_mbti.csv*

	Student Id	Gender	MBTI
1	S001	NB	ENTJ
2	S002	F	ESFP
...	...	...	...

Output: a string of the most common MBTI type. For example, the output is ENFP if it is the most common type.

```
file_path = 'coding_problems/mbti/student_mbti.csv'
```

What to do next:

```
You are on the right track by correctly setting the file path.
Now, let us focus on opening the file using this path.
```

How to do next:

```
Remember to use open() to open the file.
```

Example demonstration:

```
# here is a similar example with
# different variable names:
data_path = 'path/to/data.csv'
data_file = open(data_path, "r")
```

Solution to the next step:

```
# open the csv file using the file path

f = open(file_path, "r")
```

Figure 1: LLM Hint Factory Interface. (1) Problem description; (2) Code editor; (3) Hint Section; (3.1) Orientation Hint: the 1st level hint, informs students where they should focus; (3.2) Instrumental Hint: the 2nd level hint, informs students how to do next in concise, descriptive sentences; (3.3) Worked Example Hint: the 3rd level hint, shows students an example code snippet that is similar to the code they need to write for their next step to solve the current problem; (3.4) Bottom-Out Hint: the 4th level hint, shows students the exact code they need to write for the next step to solve the current problem; (4) Click "Be More General" button to see the previous level's hint; (5) Click "Be More Specific" to see the next level's hint; (6) Click "New Hint" to generate a new set of hint.

Participants Twelve undergraduate and graduate students were recruited from both a private research institution and a public university in the United States. These students are currently enrolled in or have only taken introductory computer science classes for non-CS majors. To verify the eligibility of the participants, we implemented a screening question designed to confirm that each participant has only taken the CS1 class using Python.

Study Procedure The think-aloud session started after checking the students' ages. During this one-hour session, participants first received a 5-minute introduction about the study consent, experiment procedure, and interface. Then participants spent 10 minutes on the pre-test, 30 minutes on the learning session, and 10 minutes on the post-test. Finally, participants received a 5-minute structured interview about their experiences using the multiple levels of hints (3.1 to 3.4 in Figure 1) in the LLM Hint Factory. The interviews aimed at understanding students' opinions on hints they received, such as whether they were comprehensible and helpful, and their suggestions for future improvement.

Experiment Materials Three programming tasks (Figure 1 demonstrates one of these tasks) about data analysis in Python were designed as practice problems for the learning session. To guarantee that the questions and their difficulty levels were suitable for our participants, we adapted them from the homework materials from the two classes from which we recruited, thus participants would be more likely to have the necessary prior knowledge and manage to complete tasks in the given time. The pre-test included eight test questions (1 open-ended, 3 multiple-choice questions, and 4 code-writing questions) about basic data extraction. The post-test contains identical programming questions as the pre-test.

5 RESULTS

5.1 Overall Hint Quality in the LLM Hint Factory

5.1.1 LLM Hint Factory can generate high-quality multiple levels of hints. We first conducted an expert evaluation of the hint quality generated during the user study. We defined a coding rubric to evaluate the quality of generated hints from 6 dimensions (table 1). The rubric was designed based on previous formative feedback design studies [4, 26] and existing works on LLM-generated programming hint evaluation [20, 23]. Following this coding rubric, two expert raters randomly sampled 20% of hint requests (resulting in 304 unique hints) generated during the user study to calculate the inter-rater reliability. Due to the high prevalence of the "Yes" label and binary classes, we applied the percent agreement approach and achieved a high agreement across dimensions. Next, two raters met and resolved the disagreements in their codes and refined the rubric. Then, one rater coded the rest of the data. Besides the manual coding on the first five criteria, the average length of each type of hint was aggregated automatically. The full evaluation results are in Table 2, and example hints fall into each category can be found in Appendix B.

Overall, all four levels of hints generated by the LLM Hint Factory are *comprehensible* and *well-targeted* to the prompt instruction at the correct level of detail. For the appropriate category, 88.16% of the orientation hints and 84.21% of the instrumental hints can point out and elaborate one problem at a time, and for the lower level hints, 75% of the worked example hints and 78.95% of the bottom-out hints can elaborate on the previous two levels of natural language hints and response with Python code pieces. However, we observed that sometimes the lower-level hints (worked example and bottom-out hints) contain code for more than one step, such as correcting the

Table 1: The expert evaluation rubric across six categories: Appropriate, Targeted, Comprehensible, Encouragement, Alignment and Length. While Alignment evaluate the interrelationship among a set of four hints, other criteria target per hint level.

Category	Definition
Appropriate - *(Yes/No)*	Is the hint a suitable next step to construct a solution given the learners' current code and its level?
Target - *(Yes/No)*	Does the generated hint target the required level?
Comprehensible - *(Yes/No)*	Is the generated hint understandable?
Encouragement - *(Yes/No)*	Does the generated hint provide any form of encouragement?
Alignment - *(0-1)*	Does the content of each level of hint align with other levels? 0: none of them align with each other; 0.25: only two of them align with each other. 0.5: pairs align with each other; 0.75: three of them align with each other; 1: all hints are well-aligned.
Length	For orientation and instrumental hints, what are the number of sentences and number of words in each hint? For worked examples and bottom-out hints, what are the lines of code in each hint?

Table 2: The expert evaluation results on Appropriate, Targeted, Comprehensible and Length for each level of hints.

Type	Appropriate	Targeted	Comprehensible	Length
Overall	81.91%	99.67%	98.36%	-
Orientational	88.16%	100%	98.68%	30 words; 2 sentences
Instructional	84.21%	98.68%	100%	29 words; 2 sentences
Worked-Example	75%	100%	100%	12 lines of code
Bottom-Out	78.95%	98.68%	93.42%	7 lines of code

import library part and the for loop in one single hint, which was considered inappropriate in this category. Regarding the *alignment* of the four levels of hints, the average alignment among 4 levels of hints in each hint request is 78.62%, which means that at least three hints in one request focus on the same problem. According to *length*, for the first two levels of natural language hints, most of them are around 30 words in two sentences. For the remaining two levels of hints (code hints), their average lengths are 12 and 7 lines respectively, with in-line code explanation comments (Table 2).

5.1.2 High-quality hints are not always helpful for students' problem-solving. We next looked at student interaction logs and recordings to further unpack the effectiveness of hints in guiding students to take correct actions. We found that, out of the expert-labeled high-quality hints, 77.78% helped students move forward on the right track in problem-solving.

So what happened to those unresolved help requests? We conducted thematic coding on students' log data and interview transcripts, and the results showed that 1.43% hints were misunderstood by the students and therefore led to negative effects. For example, after reading one worked example hint, P2 copied the code `gender = row[0]` without noticing the comment saying that, `# Assuming each row has 'gender' in the first column` , which introduced a wrong index error to their code. The remaining 27.14% sets of hints did not have an observable impact on students' progression. For these correct hints, some of them lack visual aids to draw students' attention (e.g. the line of code to `import csv` in the worked example has been missed by three participants). Some other ones were answering questions that misaligned with learners' questions in mind, making the comprehension process harder.

Students' low motivation to learn and high anxiety levels might also impact the forage of critical information in hints. These learners' urge for a direct solution might hinder their comprehension in understanding their current problem. This example demonstrated how one hint is helpful for P10 but not as helpful for P2. When they received the same orientation hint *[Now, let's focus on filtering the data for females before counting MBTI types]*, P10 skimmed the problem description again and said *"Oh, I misread the question, I should only count the most common MBTI type for females"*, and added the `if` statement in the code for the desired target. In contrast, P2 could not identify the error even after reading the hint aloud and never referred back to the problem description. Based on P2's frequent request for hints even before reading the question prompt, the low learning motivation and high anxiety on task completion may hinder the information forage from hints.

5.2 The Effectiveness of Different Levels of Programming Hints for Supporting Novices

5.2.1 Providing hints till the level of worked example can assist students properly. To understand the impact of different levels of hints, we looked at both student progress during learning sessions and their changes from pre-test to post-test. Recall that students received four levels of hints for each hint request, and the expected behavior of using a hint is to guide students toward correct changes. Therefore, we started with investigating the hint level that students needed to reach to modify the code correctly during the learning session. Among those hints labeled as appropriate by experts, in 35 (59.32%) of the instances, students were able to make correct code changes after students saw the worked-example hints; and in 5 instances, students correctly modified code after receiving bottom-out hints. In addition, while students generally revealed positive reactions (e.g. saying *"that makes sense"*) to the high-level hints, lower-level example code hints were more likely to lead to correct programming actions. For example, P24 requested a hint because she forgot the syntax of a for loop. When seeing the instrumental hint, she said *"Yeah which is exactly where I'm stuck at, but I want to* *know the exact syntax"* and immediately moved on to the example code hint. When reading through the example code in this hint, she stated, *"Oh yeah, true, looks like the correct syntax should not be the curve bracket. I should use the column."* Then they changed to the correct syntax, said *"...I wanna stop here and try to run again and see..."*, and did not proceed to the bottom-out hint.

As pre- and post-test shared the same questions, we also compared student answers from pre- to post-test to investigate how different levels of hints played a positive role in assisting students' expansion and refinement of their programming knowledge. For example, P5 could not write `import csv` in the pretest, struggled with the same item during learning session until seeing a worked example hint, and wrote this line correctly in the post-test. We found that, for those who made positive changes from pre- to post-test, all of them corrected their misconception or added new knowledge after absorbing the content in the last two levels of hints. Specifically, 75% of them were satisfied with the worked example hints and did need to get to the bottom-out level, while 25% still needed to get to the bottom-out level to make meaningful changes. Deriving from both hint request data and pre-post tests data, we deduced that hints up to the level of a worked example code can deliver proper help, primarily due to its frequent effectiveness and its balance between specificity, surpassing the high-level hints. Also, compared to bottom-out code hints, applying a worked example code hint requires higher cognitive engagement, which can encourage more meaningful learning.

5.2.2 *Worked example hints provide comprehensive help on next-step and syntax-related hint requests.*

To further examine the potential circumstances when each level of hint could be helpful, we contextualized learners' reasons to request help in programming learning as six types as shown in Table 3, following the help-seeking model [24]. After categorizing motivations of hint requests, we mapped effective levels of hints by the type of requests (Figure 2). Our results showed that, when students' confusions pertained to the progression to the next step or were syntax-related (the request types NL, NS, DS), most of their correct actions have not been taken until they saw the worked example hints. Drawing from these observations, it is recommended to furnish students with exemplar code blocks, particularly when their confusions are related to the progress to the next-step (NL, NS requests) or programming syntax (DL, DS requests).

When being asked about why the worked example hints are helpful for next-step or syntax-related requests, participants highlighted that applying the help from the worked example is straightforward, code-to-code level of help, while high-level hints require more tacit knowledge such as translating natural language into programming language or code structure visualization. P2 thought the example hint was useful as, *"when I had like, a little bit of confusion like I think I knew what I had a general idea from high-level hints what I was supposed to do but not didn't know how to execute it"*. In another case, when P8 cannot tell when to write a `return` and when not to, she did not realize there is no function definition in her code when reading the instrumental hint, *[return should be inside of a function structure]*. However, she found this part out immediately when she compared her code with the worked example hint and saw the function structure in the worked example was not in her code.

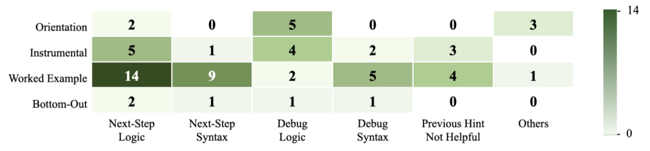


Figure 2: Number of effective level of hint for each help-seeking type. Most of requests related to next-step (NL, NS) and syntax (NS, DS) are will be resolved when learners utilized worked example hints. Most of DL requests can be answered by high-level hints.

5.2.3 *High-level hints can provide more concise help for students to debug logic errors.*

For requests about logic debugging (DL), providing high-level (orientation, instrumental) natural language guidance is often sufficient for students to effectively address and resolve errors. Multiple participants (n=4) have been reminded by the first two levels' hints to include data for females only as the problem description required, and one participant corrected the values they store for each company after seeing *[The first error is that you're not updating the total launches for each company. Both successful and unsuccessful launches should be counted in the total.]*

5.2.4 *High-level hints can cause more misunderstanding and frustration than low-level hints.*

33.3% of the students (n=4) reported their frustration when the high-level hints repeated things they already know. For example, P5 expressed as *"I got a little bit frustrated because... It was telling me something I already knew."*. Moreover, high-level hints can be easily misinterpreted due to their unspecified, descriptive nature. When P10 saw the orientation hint *[This is a really good start, you're on the right track with opening the file. Now, let's focus on reading the CSV file into a 2-D list.]*, they misunderstood the 2-D list as the output data structure of the final result, while it actually means the input data structure. Such misunderstanding was not clarified until they saw the code in the worked example hint. Additionally, high-level erroneous hints are harder to identify than low-level ones since errors in code blocks are easier to compare and more concrete. In this study, all three misleading hints that caused incorrect actions were high-level ones, while in every instance where participants received a buggy low-level hint, they were able to identify the errors.

6 LIMITATIONS

There are three major limitations to this study. The primary limitation to the generalization of these results is the small sample size of 12 programming novices from two research universities and the limited scope of knowledge used in the experiment. More generalizable findings need to be drawn from larger and more diverse participant populations with more different CS1 programming problems. Also, empirical use cases of the system can vary from experiment settings, and further classroom studies need to be conducted to further evaluate the effectiveness of hints. Lastly, the performance and cost of this system highly rely on models

Table 3: Six help-seeking types during programming problem-solving process. Adapted from the help-seeking model [24].

Type	Definition
Next-Step Logic (NL)	When a student cannot figure out the logic of the next step.
Next-Step Syntax (NS)	When a student knows the logic of the next step but do not know the syntax.
Debug Logic (DL)	When student wants to debug code with errors in logic, such as the misunderstanding of problem description.
Debug Syntax (DS)	When student wants to debug code with errors in syntax like wrong indentation.
Previous Hint Not Helpful (PNH)	When a student request for a new hint because the previous hint is not helpful.
Others	When a student asks for hints without a clear purpose, in our context, this type of request happens when learners are curious about this feature and try it out.

developed by OpenAI. Open-source, domain specific models would make this system more affordable and adjustable for researchers and educational practitioners.

7 DISCUSSION

In this work, to investigate the effectiveness of multi-level hints during programming problem-solving, we conducted a think-aloud study with 12 novices using the LLM Hint Factory. Our main finding is that, high-level hints are often insufficient to assist learners' requests, and providing code example level hint can assist students more properly. We propose the following design suggestions for our ongoing system improvement and future LLM-based learning systems.

7.1 Design Suggestion 1: Personalize the help response design under different circumstances

To avoid the rising concerns of over-reliance on LLM in learning contexts, existing GPT-powered programming tutors are deliberately designed to be natural language focused. The goal for this design is to assist students in concept understanding rather than giving the solution straight away. However, our results suggest that, in most cases, providing high-level hints alone are insufficient to help students, especially on next-step-related and syntax-related help requests. Moreover, the concise nature of the high-level hints could confuse students and cause frustration. This could lose their trust, prompting students to switch to ChatGPT for a full answer. Such findings suggest that the responses to students' help requests should be personalized based on their existing problem-solving stage and prior knowledge level.

7.2 Design Suggestion 2: Design semi-structured help-seeking interaction for flexibility and convenience

Rather than more free-style chatbot interaction, the LLM Hint Factory chose a more structured, button-triggered help-seeking interaction to make hints more accessible by reducing learners' burden of prompting quality questions, which novices are particularly bad at[15]. However, seven participants mentioned in the interview that this lack of flexibility sometimes resulted in misalignment between students' questions in mind and the problem hints tried to address, which reduced the perceived helpfulness of the hints. In the future, we intend to design a more scaffolded hint request interaction such as including a menu to allow students to select help request type or a grounded abstraction matching approach to enhance learners' question-asking technique [16]. From a learners' perspective, this interaction is flexible and convenient and potentially helps to develop skills such as debugging and planning. From the system's perspective, the hints content will be less likely to be misaligned with students, and the level shown to students can also be adapted based on the request type.

8 CONCLUSION

In this paper, we used the LLM Hint Factory to evaluate the effectiveness of multiple levels of hints on student problem-solving and learning. Our main finding is that, unlike offering high-level natural language hints alone, delivering LLM-generated hints to the level of worked example hints can provide a sufficient amount of detail in a straightforward way to students. This can facilitate their thinking and enable them to correctly proceed to the next step more effectively. The findings emphasize the need to customize responses to students' help requests from content, format, and granularity levels to accurately identify and meet student needs in future learning systems.

REFERENCES

[1] Vincent Aleven, Ido Roll, Bruce M McLaren, and Kenneth R Koedinger. 2016. Help helps, but only so much: Research on help seeking with intelligent tutoring systems. *International Journal of Artificial Intelligence in Education* 26 (2016), 205–223.

[2] John R. Anderson and Brian J. Reiser. 1985. The LISP tutor. *Byte* 10, 4 (1985), 159–175. https://www.academia.edu/download/3240480/TheLISPTutor.pdf

[3] Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. 2023. GPTutor: a ChatGPT-powered programming tool for code explanation. In *International Conference on Artificial Intelligence in Education*. Springer, Tokyo, Japan, 321–327.

[4] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*. Association for Computing Machinery, New York, NY, USA, 53–62. https://doi.org/10.1145/3160489.3160492

[5] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3411764.3445696

[6] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2017. Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *International Journal of Artificial Intelligence in Education* 27, 1 (March 2017), 65–100. https://doi.org/10.1007/s40593-015-0080-x

[7] Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2022. Using Adaptive Parsons Problems to Scaffold Write-Code Problems. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*. ACM, Lugano and Virtual Event Switzerland, 15–26. https://doi.org/10.1145/3501385.3543977

[8] Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2023. Understanding the Effects of Using Parsons Problems to Scaffold Code Writing for Students with

Varying CS Self-Efficacy Levels. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. ACM, Koli, Finland, 1–12.

[9] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–23. https://doi.org/10.1145/3544548.3580919

[10] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2023. How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. ACM, Koli, Finland, 1–12.

[11] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2019. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Transactions on Computing Education* 19, 1 (March 2019), 1–43. https://doi.org/10.1145/3231711

[12] Kenneth R. Koedinger and Vincent Aleven. 2007. Exploring the Assistance Dilemma in Experiments with Cognitive Tutors. *Educational Psychology Review* 19, 3 (Sept. 2007), 239–264. https://doi.org/10.1007/s10648-007-9049-0

[13] Harsh Kumar, Ilya Musabirov, Joseph Jay Williams, and Michael Liut. 2023. QuickTA: Exploring the Design Space of Using Large Language Models to Provide Support to Students. In *Learning Analytics and Knowledge Conference*. Learning Analytics and Knowledge Conference 2023 (LAK'23), ACM, Arlington, Texas.

[14] Sandeep Kaur Kuttal, Anita Sarma, and Gregg Rothermel. 2013. Debugging support for end user mashup programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. Association for Computing Machinery, New York, NY, USA, 1609–1618. https://doi.org/10.1145/2470654.2466213

[15] Mark Liffiton, Brad Sheese, Jaromir Savelka, and Paul Denny. 2023. CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. https://arxiv.org/abs/2308.06921v1

[16] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–31. https://doi.org/10.1145/3544548.3580817

[17] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. ACM, Toronto ON Canada, 931–937. https://doi.org/10.1145/3545945.3569785

[18] Shoaib Ahmed Malik. 2017. Revisiting and re-representing scaffolding: The two gradient model. *Cogent Education* 4, 1 (2017), 1331533.

[19] Samiha Marwan. 2021. *Investigating Best Practices in the Design of Automated Hints and Formative Feedback to Improve Students' Cognitive and Affective Outcomes - Samiha Marwan PhD Thesis - 2021*. Ph. D. Dissertation. North Carolina State University. https://doi.org/10.13140/RG.2.2.13347.32805

[20] Huy A. Nguyen, Shravya Bhat, Steven Moore, Norman Bier, and John Stamper. 2022. Towards Generalized Methods for Automatic Question Generation in Educational Domains. In *Educating for a New Future: Making Sense of Technology-Enhanced Learning Adoption (Lecture Notes in Computer Science)*, Isabel Hilliger, Pedro J. Muñoz-Merino, Tinne De Laet, Alejandro Ortega-Arranz, and Tracie Farrell (Eds.). Springer International Publishing, Cham, 272–284. https://doi.org/10.1007/978-3-031-16290-9_20

[21] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, Seattle Washington USA, 483–488. https://doi.org/10.1145/3017680.3017762

[22] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (March 2017), 37–64. https://doi.org/10.1007/s40593-015-0070-z

[23] Lianne Roest, Hieke Keuning, and Johan Jeuring. 2024. Next-Step Hint Generation for Introductory Programming Using Large Language Models. In *Proceedings of the 26th Australasian Computing Education Conference*. ACM, Melbourne, Australia, 144–153.

[24] Ido Roll, Vincent Aleven, Bruce M. McLaren, and Kenneth R. Koedinger. 2011. Improving students' help-seeking skills using metacognitive feedback in an intelligent tutoring system. *Learning and Instruction* 21, 2 (April 2011), 267–280. https://doi.org/10.1016/j.learninstruc.2010.07.004

[25] Shubham Sahai, Umair Z Ahmed, and Ben Leong. 2023. Improving the Coverage of GPT for Automated Feedback on High School Programming Assignments. In *NeurIPS'23 Workshop Generative AI for Education (GAIED)*. MIT Press, New Orleans, Louisiana, USA, 46.

[26] Valerie J. Shute. 2008. Focus on Formative Feedback. *Review of Educational Research* 78, 1 (March 2008), 153–189. https://doi.org/10.3102/0034654307313795

[27] John Stamper, Tiffany Barnes, Lorrie Lehmann, and Marvin Croy. 2008. The hint factory: Automatic generation of contextualized help for existing computer aided instruction. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track*. Springer, Montreal,Canada, 71–78.

[28] John Stamper, Michael Eagle, Tiffany Barnes, and Marvin Croy. 2013. Experimental evaluation of automatic hint generation for a logic tutor. *International Journal of Artificial Intelligence in Education* 22, 1-2 (2013), 3–17. https://content.iospress.com/articles/international-journal-of-artificial-intelligence-in-education/jai029 Publisher: IOS Press.

[29] Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '17)*. Association for Computing Machinery, New York, NY, USA, 2951–2958. https://doi.org/10.1145/3027063.3053187

[30] Anaïs Tack and Chris Piech. 2022. The AI Teacher Test: Measuring the Pedagogical Ability of Blender and GPT-3 in Educational Dialogues. In *Proceedings of the 15th International Conference on Educational Data Mining*, Antonija Mitrovic and Nigel Bosch (Eds.). International Educational Data Mining Society, Durham, United Kingdom, 522–529. https://doi.org/10.5281/zenodo.6853187

A PROMPT FOR EACH LEVEL OF HINT

A.1 Example student's current code state

A.1.1 Problem Description. You have an ice cream shop with ordering data stored in orders.csv. Your task is to find the best seller(s) and print it out.

A.1.2 Student's Code. The following is the code when the student triggered the "New Hint" button for help.

```
import csv
with open('csvIceCreamShop.csv', 'r') as file:
   reader = csv.reader(file)
   data = list(reader)
   for row in data:
        print(row)
```

A.2 Orientation Hint

A.2.1 Definition. The hint informs students where they should focus.

A.2.2 Related Prompt. The next-step hint should include at most one incomplete subgoal without any other information. The response should be within 10-50 words. Do not list the errors and subgoals directly. The rest of the hint should be consistent with the suggestion in this hint.

A.2.3 Example. This is a really good start! You've successfully read the data into a list. Now, let's focus on the next subgoal: looping over each line in the data.

A.3 Instrumental Hint

A.3.1 Definition. The hint informs students how to do next in concise, descriptive sentences.

A.3.2 Related Prompt. Describes the first error and the correct way to do it without providing a specific solution. The response should be within 10-50 words.

A.3.3 Example. The first error is that you are printing each row instead of counting the flavors. You need to loop through each row and update the count of each flavor in a dictionary.

A.4 Worked Example Hint

A.4.1 Definition. The hint shows students an example code snippet that is similar to the code they need to write for their next step to solve the current problem.

A.4.2 Related Prompt. Provides a similar example code in different scenarios and variable names similar to the bottom-out code. The number of lines and code syntax should be exactly the same as the code in the instrumental hint.

A.4.3 Example. # Imagine you have a bookstore and you want to find out which genre is the most popular. You would do something like this:

```
for row in data:
    genre = row[1] # Assuming the genre is in the 2nd column
    if genre in genre_counts:
        genre_counts[genre] += 1
    else:
        genre_counts[genre] = 1
```

A.5 Bottom-Out Hint

A.5.1 Definition. The hint shows students the exact code they need to write for the next step to solve the current problem.

A.5.2 Related Prompt. Follow the suggestion in the worked example hint, if the student's code has no error and is just incomplete, the hint code should be longer than the student's code. provides the correct version of code on the line(s) first error occurred, or the next substep this student needs to take within 5 lines of code. The code should be organized into an html code block or in an ace editor, and the explanation should be written as in-line comments of code. ALL of this hint should be written in code editor.

A.5.3 Example. # Let's focus on counting the flavors. You'll need to loop through each row and update the dictionary with the count of each flavor.

```
# Assuming the flavor is in the first column (index 0)
for row in data:
    flavor = row[0]
    if flavor in flavour_counts:
        flavour_counts[flavor] += 1
    else:
        flavour_counts[flavor] = 1
```

B EXAMPLE OF GOOD AND BAD HINTS FOR EACH CRITERIA

B.1 Appropriate

B.1.1 Student's Code.

```
import csv
file_path = 'coding_problems/icecream/csvIceCreamShop.csv'
best_sellers = []
f = open(file_path, "r")
reader = csv.reader(f)
data = list(reader)
```

B.1.2 Example Appropriate Bottom-Out Hint. When hint gives information within one step:

```
data = list(reader)
for row in data:
# iterate through all items in data
```

B.1.3 Example Not Appropriate Bottom-Out Hint. When hint gives information for more than one step:

```
data = list(reader)
for row in data:
    for flavor in row:
        if flavor in best_sellers:
            best_sellers['flavor'] += 1
        else:
            best_sellers['flavor'] = 0
```

B.2 Target

B.2.1 Example Hint Targeting on orientational level. You are moving forward! You successfully read the data, and **now try to iterate through the items in data**.

B.3 Comprehensible

B.3.1 Example Comprehensible Hint. The first error is that you're not separating the counts by gender. You need to **have separate counts for each gender to find the most common MBTI type among females**.

B.3.2 Example Not Comprehensible Hint. This is a really good start! You're on the right track with setting up a dictionary to count the MBTI types. Now, focus on **how you're updating the counts in the dictionary**.

B.4 Encouragement

B.4.1 Example Encouraging Hint. **This is a really good start, and you're on the right track with using a dictionary to count the MBTI types.** However, consider how you're currently counting MBTI types for all genders together.

B.4.2 Example Not Encouraging Hint. Consider how you're currently counting MBTI types for all genders together.

B.5 Alignment

Given a worked example hint:

```
seats = list(reader)
for row in seats:
    print(row)
```

B.5.1 Example Aligned bottom-out Hint. When the bottom-out hint provides consistent amount and content of information than the worked example hint.

```
data = list(reader)
for row in data:
    # print row to see the structure and content of each row
```

B.5.2 Example Not Aligned bottom-out Hint. When the bottom-out hint provides inconsistent amount or content of information than the worked example hint.

```
data = list(reader)
for row in data:
    for flavor in row:
        if flavor in best_sellers:
            best_sellers['flavor'] += 1
        else:
            best_sellers['flavor'] = 0
```