Integrating Natural Language Prompting Tasks in Introductory Programming Courses

Chris Kerslake
Simon Fraser University
Burnaby, British Columbia, Canada
chris.kerslake@sfu.ca

Paul Denny
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

David H. Smith IV
University of Illinois
Urbana, IL, USA
dhsmith2@illinois.edu

James Prather
Abilene Christian University
Abilene, Texas, USA
james.prather@acu.edu

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

Andrew Luxton-Reilly
University of Auckland
Auckland, New Zealand
a.luxton-reilly@auckland.ac.nz

Stephen MacNeil
Temple University
Philadelphia, PA, US
stephen.macneil@temple.edu

ABSTRACT

Introductory programming courses often emphasize mastering syntax and basic constructs before progressing to more complex and interesting programs. This bottom-up approach can be frustrating for novices, shifting the focus away from problem solving and potentially making computing less appealing to a broad range of students. The rise of generative AI for code production could partially address these issues by fostering new skills via interaction with AI models, including constructing high-level prompts and evaluating code that is automatically generated. In this experience report, we explore the inclusion of two prompt-focused activities in an introductory course, implemented across four labs in a six-week module. The first requires students to solve computational problems by writing natural language prompts, emphasizing problem-solving over syntax. The second involves students crafting prompts to generate code equivalent to provided fragments, to foster an understanding of the relationship between prompts and code. Most of the students in the course had reported finding programming difficult to learn, often citing frustrations with syntax and debugging. We found that self-reported difficulty with learning programming had a strong inverse relationship with performance on traditional programming assessments such as tests and projects, as expected. However, performance on the natural language tasks was less strongly related to self-reported difficulty, suggesting they may target different skills. Learning how to communicate with AI coding models is becoming an important skill, and natural language prompting tasks may appeal to a broad range of students.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

introductory programming, LLM, CS1, natural language prompting, prompt engineering, explain in plain English, EiPE

1 INTRODUCTION

Introductory programming courses traditionally focus on teaching students to write code in a bottom-up fashion, starting with mastering syntax and basic constructs, and gradually progressing to building more complex and interesting programs. Problem solving is often considered the most engaging aspect of programming, but the difficulties novices face with syntax, errors, and low-level code-layout issues can detract from this focus and cause frustration making computing courses less appealing to a diverse range of students [3]. Moreover, the widespread use of generative AI for producing code raises questions about when and how these tools should be introduced in introductory courses [23].

Large language models (LLMs) have shown impressive capabilities for solving computational tasks when provided with appropriate natural language prompts [6, 11, 24]. Thus, there are two emerging skills that students need to develop in this generative AI era. The first is being able to construct clear, unambiguous prompts to express desired solutions to computational tasks, and the second is understanding and evaluating the code generated by these models, to verify that it is indeed solving the intended tasks. While students may now be developing these skills independently of the curriculum, there is value in explicitly teaching students how to construct effective prompts [9].

In this experience report we describe the inclusion of two kinds of prompt-focused tasks alongside traditional activities in an introductory programming course. Both kinds of tasks involve students writing only natural language prompts for an LLM. The first task involves students solving computational tasks by writing prompts to generate code. This is a very authentic activity in today's landscape, with a focus on problem-solving rather than on code syntax. The second task involves showing students a code fragment and asking them to demonstrate their understanding of the code by crafting a prompt that generates equivalent code. The two tasks are complementary, as the first allows students to explore the relationship between computational problems and high-level prompts, and the second allows students to explore the relationship between high-level prompts and code. We are interested in understanding if the skills needed to solve these prompting tasks are distinct from those needed to be successful with traditional programming tasks.

We collected self-reported data from students on how difficult they found learning programming and why they found it difficult. We categorize students based on self-reported difficulty and then compare their performance on traditional programming assessments with their performance on the natural language prompting tasks. We also explore their perceptions of seeing these tasks integrated alongside more traditional tasks and present examples of some of the prompts that students created. Our evaluation is guided by the following overarching question: *How successful are students at natural language prompting tasks compared to more traditional programming assessments, and how does this vary by self-reported difficulty of learning to program?*

2 RELATED WORK

Developing the ability to comprehend and communicate the behaviour of code, though always considered an important set of skills for novice programmers to develop [1, 30], is necessary to work effectively with LLMs [10, 24, 27, 28]. This requires 1) the ability to describe the requirements of a problem with sufficient detail for it to be implemented in code and 2) the ability to understand the purpose of code. The former is needed as poorly constructed prompts are less likely to generate desired solutions [6] and the latter is needed to evaluate the code that is generated [5, 24].

2.1 Teaching Prompting

To develop student skills in expressing problems effectively, instructors have explored various tasks to provide students practice with prompting in formative contexts. Denny et al. [7] introduced "Prompt Problems", an activity where students are shown visual representations that illustrate specific instances of a task, asked to infer the general problem from the specific cases shown, and then provide a prompt that generates code that performs the task. The generated code is then graded based on an instructor-defined suite of test cases to determine if the generated code is functionally correct [7]. Nguyen et al. [22] evaluated similar activities where students were shown input-output pairs, asked to infer the task being performed, create a prompt that generates code with that functionality, and then evaluate if the generated code is correct. Their findings highlight that many students struggle to form successful prompts, understand generated code, and have poor mental models of generative AI, which hinders their ability to form effective prompting strategies [22].

2.2 Explain in Plain English Questions

Prompting an LLM to generate code has some similarity to the 'Explain in plain English' (EiPE) task, which is commonly used to assess student comprehension of code [21]. In these activities, students are given a code sample and asked to describe the purpose of the code [30]. EiPE activities are designed to focus on the code's high-level (abstract) purpose rather than the details of implementation (i.e., the mechanics of how it achieves the purpose). Unfortunately, novice programmers often struggle to describe the purpose of code in this way [4] suggesting greater emphasis should

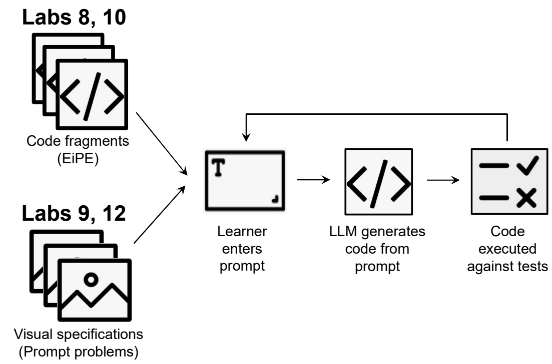

Figure 1: Natural language prompting tasks requiring the learner to enter a prompt in response to either a code fragment (EiPE) or a visual problem specification (Prompt Problem). The prompt is sent to an LLM, and the resulting code is programmatically evaluated. The EiPE and Prompt Problems were interleaved across Labs 8, 9, 10 and 12.

be placed on comprehension tasks in CS1. However, due to the difficulty of developing rubrics [13] and autograders for such questions [1, 2, 12, 14, 19], their adoption for use in formative settings has been limited.

Responses to both Prompt Problems and EiPE questions require a natural language problem specification. In the case of EiPE, this takes the form of students inferring the purpose of the code by reading it, in effect, reverse engineering the prompt that could be used to generate the given code. Using the grading approach of Smith and Zilles [28], the success of a student's prompt can be judged based on whether or not an LLM can successfully generate the desired code. Additionally, this approach both eases the difficulty of developing autograded EiPE questions and provides students with feedback via the generated code and test cases [10].

3 APPROACH

In this paper, we incorporated two kinds of natural-language prompting tasks (see Figure 1) into a course covering traditional CS1 topics and explored student perceptions and success with these tasks in comparison to more traditional programming-focused assessments (both invigilated and non-invigilated). The course was taught over a 12-week semester, although we focused on integrating these new tasks during the second half of the course (weeks 7–12). This timing allowed students to reflect on their progress throughout the first half of the course (weeks 1–6), including reporting how difficult they found programming to learn, before any exposure to the natural-language tasks.

3.1 Course Context

The course, {*Anon*}, is taught at {*Institution Anonymized*} which is a large research university in {*Country Anonymized*}. The course is designed for engineering students and is structured into two modules, each spanning 6 weeks. The first module covers typical CS1 topics, including variables, arithmetic, arrays (vectors), functions, control flow, and basic algorithms using MATLAB. The second

Table 1: List of questions and their descriptions from the four labs.

Activity	Lab	Question	Task	Description
EiPE	8	1	FindSumBetween	calculates the sum between a 'low' and 'high' value
		2	CountEvensInArray	counts the number of even values in an array
		3	LastZero	finds the position of the last occurrence of zero in an array
		4	SumPositiveValues	calculates the sum of all positive values in an array
Prompt Problems	9	1	Average	replaces each value in an array with the average of those values
		2	Sum	calculates the sum of all of the even numbers in an array
		3	Find	finds the index position of the last occurrence of zero in an array
EiPE	10	1	ReverseString	reverses a string in place
		2	FindSumOfGivenRow	calculates the sum of all values on a row in a 2D array
		3	DoesStringContainVowel	checks if a string contains a vowel
		4	DoesStringContainSubstring	checks if a string contains a substring
Prompt Problems	12	1	TwoQueens	determines if two queens attack each other on a chessboard
		2	FullQueens	determines if eight queens are placed without attacking each other
		3	LeafEater	calculates how many leaves a bug eats as it moves along a branch

module introduces the C programming language and reinforces the concepts covered in the first half of the course.

3.1.1 Student Participants. Following approval by the university's human ethics committee, data was collected from 861 students enrolled in the course. Most students have no formal programming experience, but some enter with prior experience based on their choices from high school.

3.2 Assessments and Reflection

There are three large, invigilated assessments in the course (one test for each module and a final exam), which account for 56% of the final grade in the course. The course also includes weekly programming-focused lab sessions (24% weighting) and one project (10% each) for each of the two modules, all of which are non-invigilated.

After the first module covering MATLAB, students were asked to reflect on their experience learning programming and respond with the extent to which they agree with the statement:

- *I find programming difficult.*

Responses were collected using a standard 5-step Likert-response scale from "Strongly disagree" (SD) to "Strongly agree" (SA).

In addition to this question at the beginning of the second module in the course, each lab session included an optional post-lab survey that invited students to comment on any aspect of the lab.

3.3 Natural Language Tasks

We incorporated two kinds of natural language tasks across four of the six weekly lab sessions in the second half of the course. Eight 'Explain in Plain English' (EiPE) tasks were included in Labs 8 and 10, and six Prompt Problem tasks were included in Labs 9 and 12. Table 1 summarises these 14 problems.

3.3.1 Explain in Plain English (EiPE) Questions. In the tradition of EiPE questions [13, 21], for each task, students were presented with a single function and instructions indicating that they should describe the function in plain English (see Figure 2). To prevent giving away the code's purpose, the variables were replaced with generic names and each function was named `foo`. Tasks were delivered using PrairieLearn, an open-source online assessment platform [29]. After the student description was submitted, a prompt to generate a solution meeting the description was passed to GPT-3.5. The code was evaluated against a set of test cases and then displayed to students with the results of the tests.

3.3.2 Prompt Problems. A Prompt Problem consists of a visual presentation of a computational task, to which a student must craft an LLM prompt to generate code that solves the task. In our course, we used a custom tool similar to the ones described by Denny et al. [7] and by Nguyen et al. [22]. When viewing a Prompt Problem in our tool (see Figure 3), the student sees a visual representation of the problem and enters their prompt as plain text. When their prompt is submitted, the verbatim text is sent to an LLM along with some additional prompting to guide the model to produce only code and no additional explanatory text. The generated code is executed automatically, and the test case output is shown.

4 FINDINGS

We organize the findings using students' self-reported answers to the survey question "I find programming difficult" collected at the end of Lab 7 since this provides an intuitive grouping of like-minded and potentially like-skilled students. We started with 861 students, and removed any who did not complete each of the required labs (8, 9, and 10), or who missed any of the course exams or projects, which resulted in 726 students for analysis.

4.1 Summary of Difficulty Data

A common performance pattern emerged in the course based on students' reported difficulty with programming. As shown in Table 2, students who did not report programming as difficult (Disagree or Strongly disagree) on average scored the highest on all exams and projects. In contrast, students who reported that programming was difficult (Agreed or Strongly agreed) scored the lowest on average on all exams and projects.

Table 2: LLM task attempts and success by student group. EiPE (Labs 8 & 10), Prompt Problems (Lab 9*). Difficult (Diff.) was self-reported by students during Lab 7 via a 5-point Likert scale question asking if they agreed that programming was difficult. Tries are the average number of attempts for each group. First indicates the percentage of students who completed a task on the first attempt, and Final is the percentage of students who completed the task after one or more attempts. The Projects column lists the combined averages for each groups' uninvigilated MATLAB and C programming projects.

		EiPE Problems			Prompt Problems*			Invigilated Exams			Projects
Diff.	N	Tries	First	Final	Tries	First	Final	MATLAB	C	Final	MATLAB+C
D/SD**	68	1.68	67.5%	99.6%	3.26	55.9%	100.0%	72.1%	82.4%	87.0%	93.8%
N	208	1.82	63.0%	99.7%	3.61	53.7%	100.0%	57.2%	77.0%	83.9%	92.4%
A	311	1.88	60.0%	99.4%	3.68	52.5%	100.0%	48.0%	69.8%	78.3%	87.3%
SA	139	1.89	59.9%	98.7%	4.32	45.8%	99.8%	42.8%	66.0%	73.1%	82.8%

* Unlike Labs 8, 9, and 10, Lab 12 was excluded because it was optional and less than 9% of students participated.
** Only 13 students selected Strongly disagree, so they were merged with the Disagree students to create D/SD.

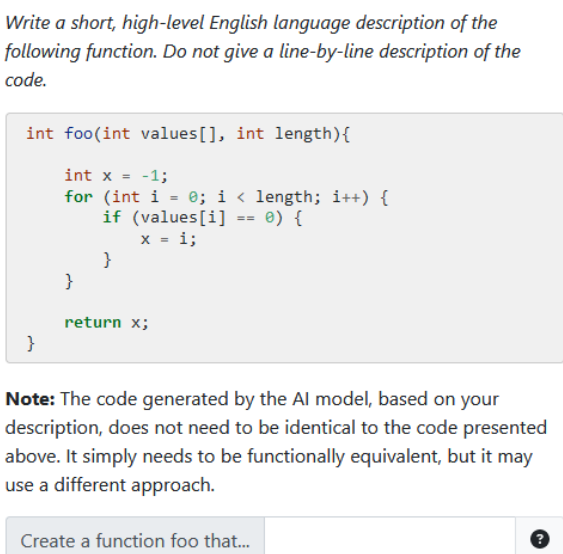

Figure 2: An example EiPE problem. The ability to highlight and copy the code is disabled to dissuade students from copying it into ChatGPT.

4.2 Student Attempt Success

For both the EiPE and Prompt Problems, we calculated the success of each student's first attempt to solve the problem (First) as well as whether they eventually were successful for each question, along with the number of attempts (Tries in Table 2). The first-attempt calculations were viewed as similar to how students would engage with problems during exams, so we compared the two to see whether students performed similarly on each or not. The final attempt (Final in Table 2) was calculated for whether students completed the tasks, similar to the course projects. The number of attempts for each type of task was calculated to determine how many attempts each of the different difficulty groups spent on the two types of problems. As detailed in Table 2, all student groups were much closer in performance on the LLM problems (Figure 5) than they were in their invigilated exam performance (Figure 4).

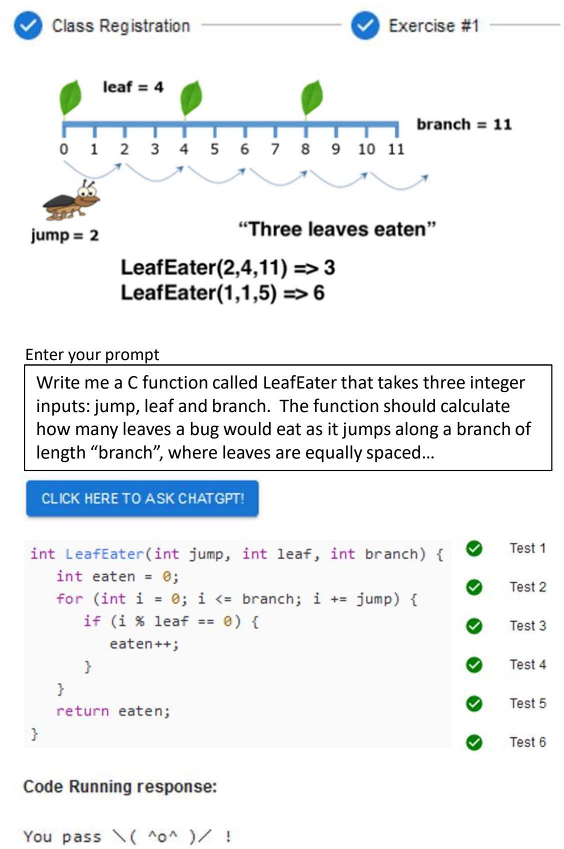

Figure 3: Interface layout for a Prompt Problem showing the LeafEater task (Lab 12, Question 3).

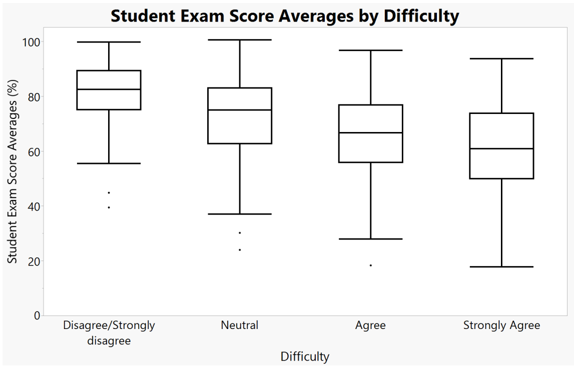

Figure 4: A comparison of student exam score averages for three invigilated exams grouped by difficulty.

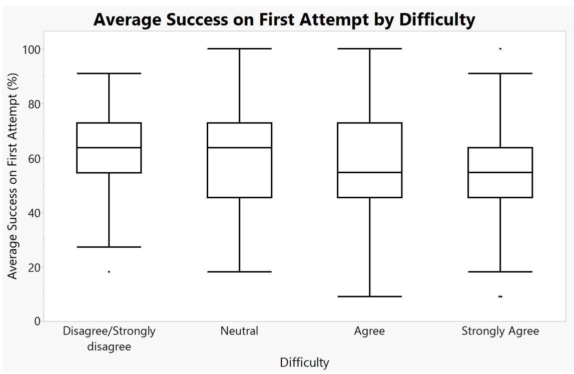

Figure 5: A comparison of students' average success rate on their first attempt to solve the LLM-related questions. The optional Lab 12 was excluded due to low participation.

4.3 Examples of Code Explanations

The code descriptions shown in Table 3 were provided by students during Lab 10 in response to code that reversed a string (ReverseString). Students provided varied prompts, but interestingly, students at both ends of the difficulty scale provided similar prompts (e.g., students 19 and 1). Additionally, some students relied on direct citation of the code as they saw it, as shown by student 105. Also of interest is that student 77 used the term "backwards" which did not result in the LLM successfully generating functionally equivalent code; in contrast to how a human might understand and interpret their meaning. However, student 13's response also used "backwards", but incorrectly. The implications are that a human reviewer might be able to understand the nuanced use of the term "backwards", but in this specific situation, the LLM could not.

4.4 Student Comments on Programming

At the end of Lab 7, students were asked to complete a feedback survey about the course. One question asked students what they enjoyed most and found most frustrating about programming. The most common enjoyment was problem-solving, and the most common frustration was debugging. As shown in Figure 6, problem-solving was the most common enjoyment for students who reported

Table 3: Student EiPE explanation examples for Lab 10's "ReverseString" question, and whether they passed test cases.

Diff.	Pass	Code Explanation (Student No.)
D/SD	Yes	"reverses a string" (19)
N	Yes	"takes one string as input and loops till length of the string - 1 and replaces str i with str of j and replaces str of j with str of i which is called a char temp, and increases i and decreases j index" (105)
A	Yes	"reverses the input string array" (1)
D/SD	No	"takes a string and turns it backwards" (77)
N	No	"writes words in a sentence backwards" (13)
SA	No	"converts a character input array to an output array of its ascii values" (106)

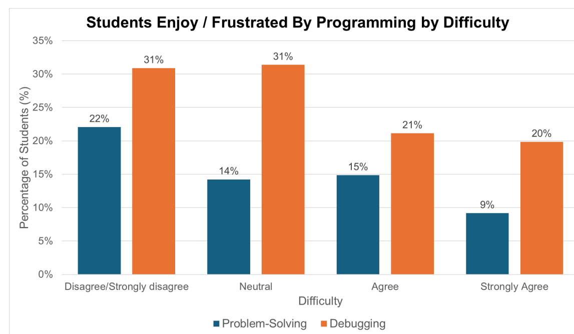

Figure 6: Percentage of students in each difficulty group who reported 'problem-solving' for enjoyment and 'debugging' for frustration.

that they enjoyed programming (D/SD; 22%) and the least reported enjoyment for students who reported that programming was difficult (SA; 9%). Additionally, debugging was reported as the most common source of frustration. Unexpectedly, students who reported the least difficulty (D/SD & N) reported it the most frequently (31%), although the other groups still reported it 20% of the time.

4.5 Student Perceptions

At the end of each LLM-related lab, students were asked to provide their thoughts on the lab and the activities. Several students provided feedback after the EiPE questions at the end of Labs 8 and 10, shown below. Overall, students were predominantly positive about the experience, with a single student calling the task "gimmicky" and less effective at assessing their performance than traditional code-writing tasks.

- *Positive: "I thought the code comprehension task was good, because it encourages understanding of code logic without the pressure of having to write code or debug. It also helps to improve my ability to communicate what a piece of code does* *by forcing me to write clear and concise explanations of code that can be easily understood." (402, Strongly Agree; Difficult)*

- *Positive: "I found the starting of the code comprehension task really difficult. This was because as I was doing it I was over complicating it each time. However with more attempts and practice it became easier. I really liked this task as it helped me on how to understand and condense my understanding with basic concepts in coding." (329, Agree; Difficult)*
- *Positive: "[EiPE problems] also enlightened me on the different solutions that could be available to solve a specific task" (510, Neutral)*
- *Negative: "Dislike the [EiPE] tasks. Feels gimmicky. It's to me as if I saw this new AI-powered tool, and I'm using it just because I learnt about it [rather than for the tool's merits]. I don't think it proves any student performance, and the code-writing based lab tasks prove our understanding much more." (350, Disagree/Strongly disagree; Easy)*

5 DISCUSSION AND TAKEAWAYS

As noted by Porter & Zingaro [23], the introduction of LLMs suggests re-evaluating how we teach introductory programming courses. Additionally, with increased class sizes, the need to automate learning at scale highlights the importance of identifying techniques and tools that can help students assess and improve their code comprehension abilities. However, they also note that LLMs do not replace the need to understand what the code does when generated, nor does it remove the need for being able to explain the problem sufficiently to produce the desired results. The tasks we have analyzed in this work – Prompt Problems and EiPE questions – aim to teach students these skills explicitly via generative AI, and provide automated feedback that can support their use at scale.

Recent work has shown that the introduction of generative AI into computing classrooms is negatively impacting student programming skills like code writing and debugging. Jost et al. found a significant negative correlation between increased use of LLMs for coding tasks and lower critical thinking skills as well as a decrease in final grades [15]. Prather et al. found that generative AI code completion tools like Copilot and ChatGPT can benefit some students who are already confident in their programming abilities, but that it can be directly harmful to the programming problem solving ability of students who are not [26]. Students who already have difficulty with programming, like many of the ones we examined in this study, seem to be the most poised to over-rely on generative AI [20] and its harms could be compounded upon them.

Our findings suggest that natural language tasks could 'bridge the gap' between students who struggle with traditional assessments and those who do not, which could engage a broader range of students and possibly address the harmful impacts of generative AI on this group. Additionally, most students found the natural language programming tasks positive, indicating it is not useless for those who do not struggle. Most students required multiple attempts to craft a prompt that correctly solved their given task, similar to prior work [7, 22], which provides additional support for explicitly teaching students 'prompt engineering.' Previous work has found that this kind of iterative learning while solving Prompt Problems could support the development of metacognitive skills in novice programmers [25], which is possibly why these tasks can help address the potential negative impacts of generative AI.

There are many possible explanations for the students' performance being more similar on the natural language tasks than traditional code writing tasks. One is that success on the natural language tasks does not rely on mastery of low-level syntax, which can be hard for novices to get right [8, 18]. Another is that the natural language tasks are simply easier, and therefore less likely to differentiate between students who are more or less confident with programming. However, typical success rates on the first attempts at both problem types (usually in the 50% to 60% range) suggest that the tasks are not trivial to solve. Finally, a further possible explanation is that these tasks target different skills altogether compared to traditional assessments. In traditional code writing tasks, students can, for example, tinker their way to a solution [16], unable to explain the code they have written [17], which signals that they do not understand it. In the natural language tasks, the student's goal is at a higher level of abstraction – they do not need to think about how to write the code to solve the problem but instead describe how the code works at a high level (EiPE), or describe the functionality of the code in natural language (Prompt Problems).

The finding that most students found the natural language tasks positive is similar to findings in prior work. Nguyen et al. [22] found that most students would use a natural language programming tool again if it were available, and Denny et al. [7] reported that most students found similar tasks educational.

6 CONCLUSION

In this work, we describe our experience including two kinds of natural language prompting tasks alongside traditional assessments in an introductory programming course. We observed a weak relationship between performance on these tasks and more traditional programming assessments, suggesting that these new tasks may assess a different set of competencies. We also collected self-reported data from students on the difficulty they experienced learning programming. Interestingly, self-reported difficulty was very strongly related to performance on tests, exams, and programming projects, as would be expected, but this was not the case for the natural language tasks. In other words, students with less prior experience or those who find traditional programming challenging appear to perform relatively well on these tasks, potentially reducing the advantage experienced students typically have. We also found that students appreciated these types of tasks and recognized the importance of learning about AI and its applications in programming.

REFERENCES

[1] Sushmita Azad. 2020. *Lessons learnt developing and deploying grading mechanisms for EiPE code-reading questions in CS1 classes.* Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[2] Sushmita Azad, Binglin Chen, Maxwell Fowler, Matthew West, and Craig Zilles. 2020. Strategies for Deploying Unreliable AI Graders in High-Transparency High-Stakes Exams. In *21st International Conference, AIED 2020, Ifrane, Morocco, July 6–10, 2020, Proceedings, Part I*, Ig Ibert Bittencourt, Mutlu Cukurova, Kasia Muldner, Rose Luckin, and Eva Millán (Eds.), Vol. LNAI 12163. Springer Cham, Cham, Switzerland, 16–28. https://doi.org/10.1007/978-3-030-52237-7_2

[3] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and*

Technology in Computer Science Education (Aberdeen, Scotland, UK) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[4] Jeffrey Bonar and Elliot Soloway. 1983. Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '83)*. Association for Computing Machinery, New York, NY, USA, 10–13. https://doi.org/10.1145/567067.567069

[5] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software* 203 (Sept. 2023), 111734. https://doi.org/10.1016/j.jss.2023.111734

[6] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. 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 (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1136–1142. https://doi.org/10.1145/3545945.3569823

[7] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 296–302. https://doi.org/10.1145/3626252.3630909

[8] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. Association for Computing Machinery, New York, NY, USA, 75–80. https://doi.org/10.1145/2325296.2325318

[9] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan 2024), 56–67. https://doi.org/10.1145/3624720

[10] Paul Denny, David H. Smith, Max Fowler, James Prather, Brett A. Becker, and Juho Leinonen. 2024. Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 283–289. https://doi.org/10.1145/3649217.3653587

[11] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Proceedings of the 24th Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[12] Max Fowler, Binglin Chen, Sushmita Azad, Matthew West, and Craig Zilles. 2021. Autograding "Explain in Plain English" questions using NLP. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1163–1169. https://doi.org/10.1145/3408877.3432539

[13] Max Fowler, Binglin Chen, and Craig Zilles. 2021. How should we 'Explain in plain English'? Voices from the Community. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 69–80. https://doi.org/10.1145/3446871.3469738

[14] Silas Hsu, Tiffany Wenting Li, Zhilin Zhang, Max Fowler, Craig Zilles, and Karrie Karahalios. 2021. Attitudes Surrounding an Imperfect AI Autograder. 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/10.1145/3411764.3445424

[15] Gregor Jošt, Viktor Taneski, and Sašo Karakatič. 2024. The Impact of Large Language Models on Programming Education and Student Learning Outcomes. *Applied Sciences* 14, 10 (2024), 4115.

[16] Teemu Lehtinen, Lassi Haaranen, and Juho Leinonen. 2023. Automated Questionnaires About Students' JavaScript Programs: Towards Gauging Novice Programming Processes. In *Proceedings of the 25th Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE '23)*. Association for Computing Machinery, New York, NY, USA, 49–58. https://doi.org/10.1145/3576123.3576129

[17] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. 2021. Students Struggle to Explain Their Own Program Code. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/3430665.3456322

[18] Antti Leinonen, Henrik Nygren, Nea Pirttinen, Arto Hellas, and Juho Leinonen. 2019. Exploring the Applicability of Simple Syntax Writing Practice for Learning Programming. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Assoc. for Computing Machinery, New York, NY, USA, 84–90. https://doi.org/10.1145/3287324.3287378

[19] Tiffany Wenting Li, Silas Hsu, Max Fowler, Zhilin Zhang, Craig Zilles, and Karrie Karahalios. 2023. Am I Wrong, or Is the Autograder Wrong? Effects of AI Grading Mistakes on Learning. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1 (ICER '23, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 159–176. https://doi.org/10.1145/3568813.3600124

[20] Lauren E. Margulieux, James Prather, Brent N. Reeves, Brett A. Becker, Gozde Cetin Uzun, Dastyni Loksa, Juho Leinonen, and Paul Denny. 2024. Self-Regulation, Self-Efficacy, and Fear of Failure Interactions with How Novices Use LLMs to Solve Programming Problems. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024)*. ACM, New York, NY, USA, 276–282. https://doi.org/10.1145/3649217.3653621

[21] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in plain English' questions: implications for teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 385–390. https://doi.org/10.1145/2157136.2157249

[22] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–26. https://doi.org/10.1145/3613904.3642706

[23] Leo Porter and Daniel Zingaro. 2024. *Learn AI-assisted Python programming: with GitHub Copilot and ChatGPT* (first edition ed.). Manning Publications, Shelter Island, New York.

[24] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots Are Here: Navigating the Generative AI Revolution in Computing Education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '23)*. Association for Computing Machinery, New York, NY, USA, 108–159. https://doi.org/10.1145/3623762.3633499

[25] James Prather, Paul Denny, Juho Leinonen, David H Smith IV, Brent N Reeves, Stephen MacNeil, Brett A Becker, Andrew Luxton-Reilly, Thezyrie Amarouche, and Bailey Kimmel. 2024. Interactions with Prompt Problems: A New Way to Teach Programming with Large Language Models. *arXiv preprint arXiv:2401.10759* (2024), 30 pages. https://doi.org/10.48550/arXiv.2401.10759

[26] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1 (ICER '24, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 469–486. https://doi.org/10.1145/3632620.3671116

[27] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. 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* (Lugano and Virtual Event, Switzerland) *(ICER '22)*. Association for Computing Machinery, New York, NY, USA, 27–43. https://doi.org/10.1145/3501385.3543957

[28] David H. Smith and Craig Zilles. 2024. Code Generation Based Grading: Evaluating an Auto-grading Mechanism for "Explain-in-Plain-English" Questions. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 171–177. https://doi.org/10.1145/3649217.3653582

[29] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. In *2015 ASEE Annual Conference & Exposition*. American Society for Engineering Education (ASEE), Seattle, WA, USA, 26.1238.1–26.1238.14. https://doi.org/10.18260/p.24575 ISSN: 2153-5965.

[30] J. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. K. A. Kumar, and C. Prasad. 2006. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education*, Denise Tolhurst and Samuel Mann (Eds.), Vol. 52. Australian Computer Society, Inc., Hobart, Australia, 243–252. https://hdl.handle.net/10292/15405