Non-Expert Programmers in the Generative AI Future

Molly Q Feldman*
Oberlin College
USA
mfeldman@oberlin.edu

Carolyn Jane Anderson*
Wellesley College
USA
carolyn.anderson@wellesley.edu

ABSTRACT

Generative AI is rapidly transforming the practice of programming. At the same time, our understanding of who writes programs, for what purposes, and how they program, has been evolving. By facilitating natural-language-to-code interactions, large language models for code have the potential to open up programming work to a broader range of workers. While existing work finds productivity benefits for expert programmers, interactions with non-experts are less well-studied. In this paper, we consider the future of programming for non-experts through a controlled study of 67 non-programmers. Our study reveals multiple barriers to effective use of large language models of code for non-experts, including several aspects of technical communication. Comparing our results to a prior study of beginning programmers illuminates the ways in which a traditional introductory programming class does and does not equip students to effectively work with generative AI. Drawing on our empirical findings, we lay out a vision for how to empower non-expert programmers to leverage generative AI for a more equitable future of programming.

CCS CONCEPTS

• **Human-centered computing** → **User studies**; • **Social and professional topics** → **Computing education**; • **Computing methodologies** → *Artificial intelligence*; *Machine learning*; • **Software and its engineering**;

ACM Reference Format:
Molly Q Feldman and Carolyn Jane Anderson. 2024. Non-Expert Programmers in the Generative AI Future. In *Proceedings of the 3rd Annual Meeting of the Symposium on Human-Computer Interaction for Work (CHIWORK '24), June 25–27, 2024, Newcastle upon Tyne, United Kingdom.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3663384.3663393

1 INTRODUCTION

Large language models for code, or *Code LLMs*, promise to reshape how programming is performed across the economy. Code LLMs are large neural networks trained to perform next token prediction on a mixed corpus of natural language and code; as a result, they can be used for a range of text generation tasks that involve code and natural language, such as code completion and program synthesis [56].

Code LLMs have the potential to make programmers more efficient [7, 11, 73, 82] and to broaden access to programming by supporting natural-language-to-code interactions. In this paper, we consider what the advent of Code LLMs means for *non-expert* programming workers. We investigate the current usability of Code LLMs for non-experts, consider the impact of traditional computer science (CS) education on prompting ability, and envision an educational approach for equipping non-experts to work effectively with Code LLMs.

Code LLMs promise to reduce the technical barrier to performing coding tasks: instead of needing to know how to write a program by hand, a user can simply describe the desired behavior to the Code LLM, which then generates the code. As a result, Code LLMs could empower *non-expert programmers*: workers who 1) lack substantial formal education in CS; 2) do not consider programming their primary job responsibility; but 3) perform programming tasks at work to facilitate their core duties. To give some concrete examples, non-expert programmers include consultants who write code to compute statistics; scientists who write code to interface with lab instruments; or even restaurant managers seeking to customize reservation systems.

Non-expert programmers exhibit a spectrum of programming skill levels. Figure 1 provides one characterization of such programmers as part of a programming "food chain." The most proficient we call **Code Writers**, or workers who can write programs independently. More commonly, however, non-experts are **Code Adaptors**: they cannot write code from scratch, but they can understand and adapt code from various sources [46]. Even more workers are **Code Runners**: they learn to run pre-written code by demonstration. They do not have the skills to resolve problems they encounter with pre-written code, making them beholden to others higher up in the food chain [16].

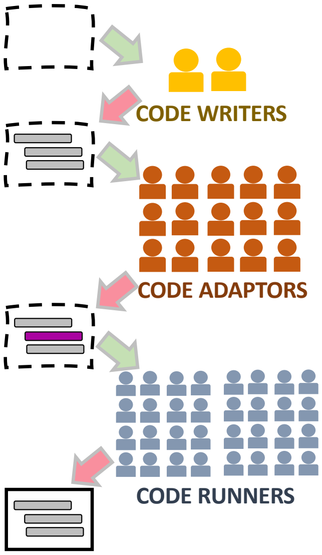
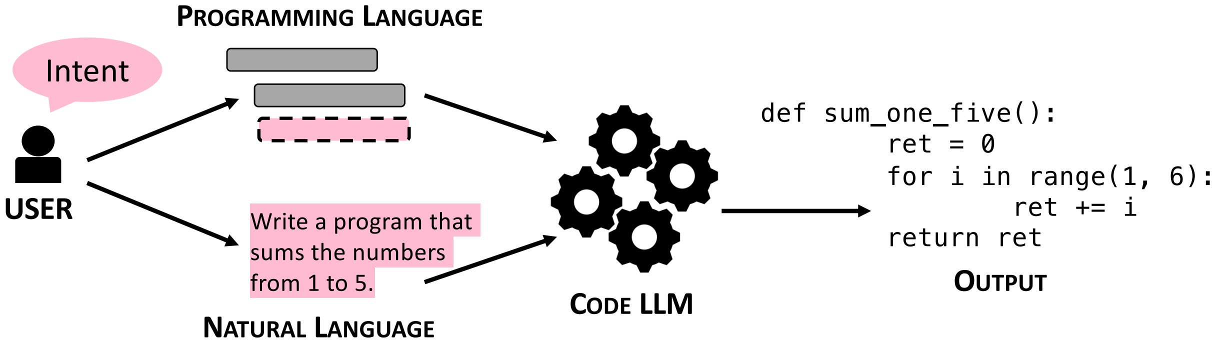


Figure 1: Worker skills in the programming food chain.

[*] Equal contribution

Conditionally Accepted to CHIWORK '24, June 25–27, 2024, Newcastle upon Tyne, United Kingdom



```
def sum_one_five():
    ret = 0
    for i in range(1, 6):
        ret += i
    return ret
```

Figure 2: An example Code LLM interaction. Most Code LLMs allow users to specify prompts in programming language, natural language, or a combination. The prompt is then passed to the Code LLM, which will attempt to generate output. If the model output is not correct, or does not match the user's intent, the user can modify their prompt and then try again.

We believe that Code LLMs could *flatten the programming food chain*. Code Adaptors could use them to generate their own programs, since they are familiar with the terminology used to talk about code and have skills related to evaluating and adapting code. Code Runners can benefit as well: LLM-generated documentation could help them understand code, allowing them to level up to become Code Adaptors or even produce code themselves,[1] though there may also be concerns about the accuracy of model-generated documentation.

However, achieving this future rests on the assumption that non-experts can communicate effectively with Code LLMs. This may be challenging for a number of reasons. Working with a Code LLM is a multi-step process, involving forming an intent; describing the intent in code or natural language; reviewing generated code for correctness; and iteratively revising the prompt to regenerate, or manually editing the code to fix any issues (Figure 2).

When a non-expert programmer describes their intent in natural language, they may not use the same words that a professional software engineer would. This is an issue, because Code LLMs are typically trained on professionally-produced code [44]. Moreover, Code Adaptors and Code Runners may lack the ability to judge whether model-generated code meets their needs. While an expert programmer would be able to manually correct portions of the generated code, a Code Runner can only choose to discard the code and start over. This gap between professional practice and non-expert ability mirrors decades of challenges providing training and feedback to Code Writers [12, 40, 74].

A key question of the future of work with Code LLMs therefore is how they will affect non-expert programmers. Will Code LLMs bridge the skills gap between experts and non-experts? Or will non-experts fall even farther behind as experts are able to leverage the power of generative AI that remains inaccessible to others?

In this paper, we consider how Code LLMs will impact non-expert programmers for both better and worse. We focus specifically on Code Adaptors and Code Runners, since they differ the most from the professional programmers in previous studies of Code LLMs in the workplace. We investigate the ways in which current Code LLMs do and do not meet their needs via a study of how non-programmers interact with Code LLMs. We consider the ways in which traditional Introduction to Computer Science (CS1) courses do and do not equip non-expert programmers for the age of generative AI, and envision a new approach to training non-experts as **Code Builders**, that will equip them to effectively work with Code LLMs, ensuring that Code LLMs reduce, rather than increase, the skills gaps between expert and non-expert programming workers.

We draw on existing studies of Code LLM use by beginning programmers in secondary school [35, 36], in CS1 courses [18, 65, 66], and after CS1 [32, 60], which have revealed barriers related to technical communication, understanding generated code, and the unpredictable nature of Code LLMs. We take the experimental design from a previous study of beginning programmers [60] and adapt it for a non-programmer population: university students with *no* formal programming experience. This allows us to directly compare non-programmers to students with one semester of traditional CS education.

Our exploration[2] considers questions relevant to numerous aspects of non-expert use of Code LLMs. First, we are interested in establishing a *baseline*: how effective are current models for non-programmers and what key skills do non-experts need to use such models effectively? We are then interested in observing the *effect of traditional CS1* on participant performance, highlighting the ways in which CS1 equips students to engage more successfully with Code LLMs. Finally, we *imagine the future* by proposing the set of key skills non-expert programmers need to succeed in the era of generative AI. We conclude with a set of recommendations for the various stakeholders involved: Code LLM model developers, Code LLM interface designers, and educators.

2 CODE LLMS AS A FUTURE OF WORK TECHNOLOGY

Our work focuses on envisioning a future for non-expert programmers empowered by Code LLMs. The educational background of non-expert programmers can vary widely. Some may have learned programming in a class setting, such as a CS1 course or a statistics

[1] See Guo [28] for discussion of this possibility within the natural sciences context.

[2] The data collected as part of this study is publicly available at https://doi.org/10.17605/OSF.IO/MXH35

course with required R programming. Others engage in self-study through an online program or as part of their job training. Our definition of *non-expert* encompasses varying backgrounds, but excludes anyone with the equivalent of a computing major or minor.

Non-expert programmers from various fields are beginning to experiment with Code LLMs and to share their experiences. For instance, on the r/consulting Reddit, a forum for consultants, there are many threads about generative AI. In discussions of Code LLMs, a recurrent theme is that the commenters who report efficiency gains typically also report prior proficiency in programming. For instance, commenter *LaOnionLaUnion* writes, "I do a lot of data wrangling from Excel and CSV files. It's a lot faster for me to code that in Python with [a] LLM than to do the same things inside Excel even one let alone many times. Granted, I understand how to code so it is not hard for me to troubleshoot, spot problems, or direct it towards better results."[3]

This quotation highlights both the promise and peril of generative AI for non-expert programmers. It can speed up routine tasks and make it easier to work with different programming frameworks or languages. However, working effectively with Code LLMs also requires the ability to debug, to evaluate code quality, and to craft prompts that the model understands. As a result, Code LLMs could actually lead to a growing productivity gap between expert and non-expert programmers, if only experienced programmers are able to effectively leverage them.

Although Code LLMs have the potential to reshape many workplaces, interacting with Code LLMs is still a novel experience for most workers. This makes studying user interactions with Code LLMs a key research area for the future of work, echoing previous iterations of studying user intent with novel programming technologies [47]. Specifically, our collective understanding of how to study such interactions builds on insights from work on program synthesis [23, 27, 31, 58], literate programming [39, 42], as well as other areas [43, 71]. Below we summarize specifically what is known about Code LLM interactions with different users to place our results, and recommendations, in context.

Expert Interactions. Given our focus on Code Adaptors and Code Runners, we study only users prompting the Code LLM with natural language prompts. However, existing work on code prompting sheds light on how experts work with Code LLMs.

An early study of Code LLM-user interaction is Xu et al. [80], which looked at 31 experienced programmers interacting with a custom-built programming assistant in PyCharm. Much subsequent work uses Github Copilot (2021), one of the first publicly available Code LLMs, as the backend. Copilot notably allows both natural-language-to-code and code-to-code interactions. Vaithilingam et al. [73]'s study of 24 experienced programmers showed that participants found Copilot useful for basic coding tasks and to replace search behaviors. In three studies of Copilot interactions, Bird et al. [11] find an increase in programmer productivity and speed, at the cost of program understanding.

Other work has focused on developing theories of Copilot-expert interaction. Barke et al. [7] develop a grounded theory for Copilot interaction based on a study of 20 expert participants. They find two main interaction modes, acceleration and exploration, which allow for programmers to make progress on the given tasks. A more recent, at-scale survey of 410 developers [53] confirms and expands Barke et al. [7]'s findings. Of particular relevance to our work is their finding that participants requested more natural language prompting as part of their existing Code LLM tools. When describing why users prefer natural language/chat, they report "because providing clear explanations, often in natural language, was the most cited strategy to having AI programming assistants produce the best output" [53].

Taken together, these works suggest that Code LLMs can have a positive effect on expert interaction. However, these studies operate on the assumption that participants have enough knowledge to prompt the model and understand its output. Our work explores how the lack of these skills may inhibit non-experts and asks how we can structure skill acquisition so that non-experts also experience productivity gains from Code LLMs.

Non-Expert Interactions. When considering non-expert programmers, we draw upon existing work studying Code LLMs in educational spaces. Prather et al. [66] study 19 students in an Introduction to Computer Science (CS1) course working with Copilot on a Minesweeper final project in C++. They find two behaviors unique to the beginning student population, in comparison to experts: the phenomenon of attempting to coerce Copilot to generate code (shepherding) and a back-and-forth cycle of hesitantly accepting, then deleting, presented code (drifting). Jayagopal et al. [32] report on an observational study of how second-semester CS students interact with Copilot and four programming tools, highlighting important design decisions for user interfaces. Our work complements these findings: we look at participants without any formal CS education (non-programmers), who are more akin to the larger population of non-experts.

Turning to natural language prompting, researchers have considered user interactions with Code LLMs via written descriptions as well. Kazemitabaar et al. [35, 36] study secondary school students with and without access to Copilot in an online learning environment, finding that access to the Code LLM was not associated with a decline in student learning. However, tasks were presented to students in natural language, which led students to copy/paste the expert-written description into the model around half of the time. As a result, it is hard to use their data to understand whether students would be able to write effective prompts on their own.

Other studies address this limitation by adopting different task presentations. Denny et al. [18] and Prather et al. [65] used a variety of visual mechanisms to present problems (Prompt Problems), including images of terminal output and graphics of judges holding score cards. They found this effective in a deployment during the second week of a CS1 course. Nguyen et al. [60] presented problems through input-output examples in their study of 120 students who had completed CS1. We adapt and extend their design to students without formal CS education in our study.

Multi-turn interactions. Our work focuses on single-turn interactions: participants provide a natural language description, which is used to prompt the model, and if the interaction is unsuccessful, the user edits their prompt and tries again. However, some models also support multi-turn interactions, where the model has access

[3] https://www.reddit.com/r/consulting/comments/17stayn/do_you_actually_believe_that_chatgpt_made_bcg

to the entire history of interactions with the user while generating its response. There is little existing work on multi-turn interactions, since they are not easily studied in controlled experiments. Ross et al. [70] is one exception; they conduct a study of 42 experienced programmers interacting with a conversational Code LLM-based assistant and find that it is positively received. A rare case of domain-specific work is Chen et al. [14], which looks at experts and novices using an LLM to aid programming in a language designed for agent-based modeling. They find a "knowledge gap" between experts and novices that mirrors the gap we observe between beginners and non-programmers.

3 CHALLENGES WITH STUDYING CODE LLM INTERACTIONS

When developing studies on non-expert interactions with Code LLMs, there are some key design decisions. Using Code LLMs involves several processes: forming an intent, expressing the intent as a natural language prompt, interpreting model output, and iterating. All of these steps are more challenging for non-programmers than for experienced programmers.

In a controlled environment, where the programming tasks have been pre-selected, there are multiple options for presenting a task. Although a natural language presentation may seem natural, it has a substantial drawback: as previous work has found [35], when participants are shown natural language descriptions, they tend to copy/paste instead of writing their own prompts. An alternative is to present tasks through examples [18, 60] (Figure 3). Specifically, participants are given a set of input/output pairs illustrating what the program should do.

Another design decision is the interface to the Code LLM. Code LLMs are being integrated into many popular IDEs,[4] such as Microsoft's VSCode. However, non-expert programmers may not be familiar with IDEs and may find the variety of features overwhelming; even the second-semester CS students in Jayagopal et al. [32] struggled with the Copilot/VSCode integration. This is why many CS1 classes use simple text editors or IDEs designed specifically for teaching. Therefore, though studies of expert programmers often use IDE-integrated Code LLMs, studies of non-experts often use simple platforms that restrict how users interact with the model.

One of the most challenging aspects of the Code LLM task for non-experts is evaluating the generated code for correctness, style, etc. In studies of programmer interactions with Code LLMs, there is a tension between studying prompt writing and code evaluation. If non-experts make mistakes in judging the generated code, they may move on from a task without actually completing it; however, providing feedback on the generated code to guide the participant removes the opportunity to study the code-judging step of the full Code LLM interaction loop. Previous work has shown that determining correctness is a challenging task for expert programmers [73]. For this reason, we choose to focus our study on how non-programmers engage in the prompt-writing and prompt-modifying steps of the process, and take care of the code-judging step for our participants.

4 HOW DO NON-PROGRAMMERS INTERACT WITH CODE LLMS?

To illustrate the challenges that non-experts currently face in interacting with Code LLMs, we present a study exploring how non-programmers work with Code LLMs to solve problems drawn from traditional Introduction to Computer Science (CS1) courses. We replicate the experimental methodology of Nguyen et al. [60], but with a non-programmer population. By adopting the same experimental paradigm, we can compare our populations and explore how CS1 does and does not prepare students to effectively use Code LLMs, in addition to understanding how non-programmers approach the natural-language-to-code task.

4.1 Experimental Design

Participant Recruitment and Screening. We recruited 67 undergraduates from two primarily undergraduate institutions, Oberlin College and Wellesley College, in late 2023.[5] Students were eligible if they had no prior programming experience. Recruitment was done by email, announcements in humanities classrooms, and flyers. The study took around 45 minutes; participants received a $30 USD gift card. The study was approved by the IRB at Wellesley College under a reliance agreement with Oberlin College.

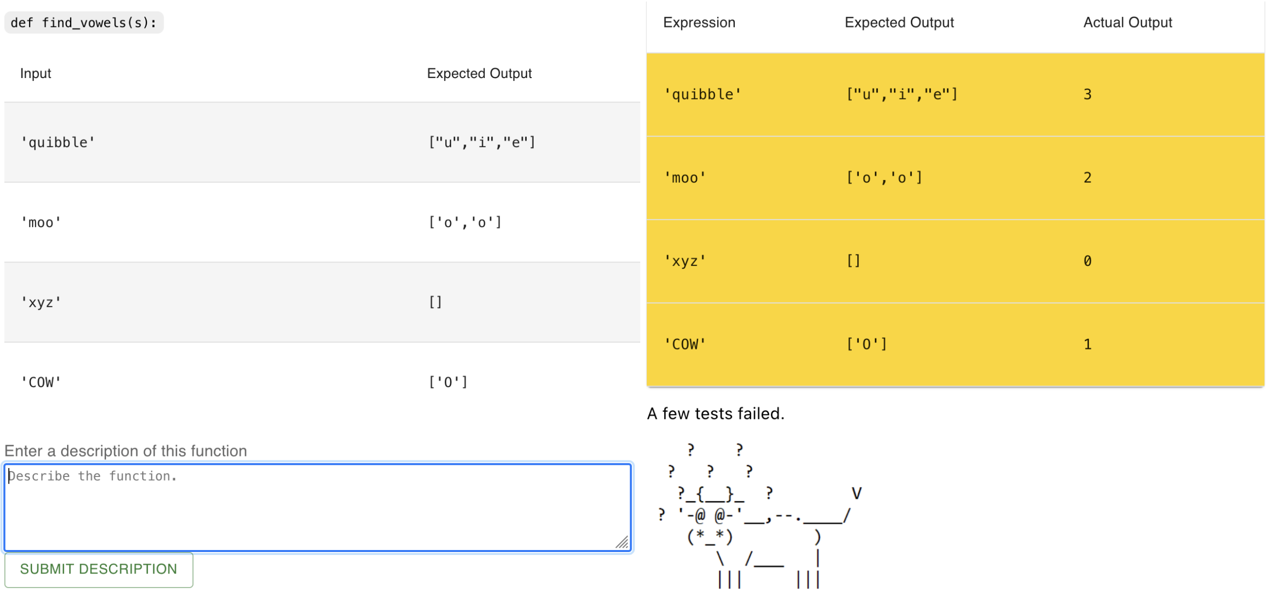
Our study targeted students with no prior programming experience. However, many undergraduates today have some limited exposure to programming through extracurricular activities or non-CS classes in secondary school. We screened each participant by asking about various types of programming experiences. We excluded students who had taken high school or college courses with weekly programming tasks, including non-CS courses like statistics; had done a summer internship involving programming; or had participated in a recurring extracurricular programming class or club, like a robotics club. We included participants who had taken courses with one or two programming activities, for instance, a middle school math course with a week of Scratch exercises. We did not filter participants based on previous knowledge of LLMs or Code LLMs; rather, we asked whether they had heard of several popular models in the post-task survey to gauge familiarity.

Materials. We used a subset of the programming tasks from Nguyen et al. [60], removing their complex data structure and math categories. Four problem categories remained: strings, lists, conditionals, and loops. The Nguyen et al. [60] problems were sourced from CS1 courses so that they would be at an appropriate level for their participants, who had taken CS1. It is harder to determine the right difficulty for non-programmers, since they have more varied mathematical backgrounds; however, using the same problems lets us draw direct comparisons between the two studies.

Model. To facilitate comparison, we used the same Code LLM as Nguyen et al. [60]: OpenAI's Codex (`code-davinci-002`) model. At the time of the study, Codex was used in the free version of ChatGPT and outperformed the model that was in use for GitHub Copilot's inline completions [81]. Moreover, larger models like

[4] An *integrative development environment*, or IDE, is a program for writing software that incorporates features beyond simple text-editing, such as syntax highlighting, auto-complete, and debugging tools.

[5] These institutions are US small liberal arts colleges (~2,500 total students), where most students are studying for Bachelors of Arts degrees across majors in the humanities, social sciences, and natural sciences



```
def find_vowels(s):
```

Input	Expected Output
'quibble'	["u","i","e"]
'moo'	['o','o']
'xyz'	[]
'COW'	['O']

Enter a description of this function

Describe the function.

Expression	Expected Output	Actual Output
'quibble'	["u","i","e"]	3
'moo'	['o','o']	2
'xyz'	[]	0
'COW'	['O']	1

A few tests failed.

```
   ?    ?
 ?   ?   ?
  ?_{__}_  ?        V
? '-@ @-'__,--.____/
   (*_*)        )
      \  /___   |
      |||     |||
```

Figure 3: How the natural-language-to-code task is presented to participants. On the left is a (tutorial) task described by a set of input-output examples along with the function signature. Participants write their description in the text box and then press Submit. On the right is part of the result screen: in this case, 4 tests failed and are highlighted. Not pictured: the generated code and the buttons to re-try the problem or move on.

GPT-4 [64] have high latency, which frustrates expert [59] and non-expert users [37]. Using Codex also reflects the reality of non-experts, since state-of-the-art models tend to have higher cost or other barriers to use.

Procedure. Our study consisted of a short tutorial, followed by four Python natural-language-to-code tasks, and a survey. Our experimental procedure was based on Nguyen et al. [60] with some key modifications. Although we use the same web-based platform, we created a more substantial tutorial for non-programmers with simpler problems (Figure 3). We reduced the number of tasks from 8 to 4, removing the more difficult timed tasks, and removed questions asking students to rate the generated code. Moreover, our study was conducted in-person rather than remotely: we used computer classrooms where students worked individually on separate computers. Experimenters were present to answer questions or resolve technical difficulties, but otherwise did not intervene. This meant we adapted both the task debrief and a subset of the interview questions from Nguyen et al. [60] into survey questions.

Participants were screened for prior programming experience and led to a computer. After completing an informed consent form, they worked through a short tutorial introducing them to the platform and the Code LLM tool, which was presented using a cow avatar named Charlie (Figure 3). The main study consisted of four tasks assigned randomly via a latin square design (similar to that done in Nguyen et al. [60]), each presented as a set of input/output examples (test cases). Students wrote a description of the problem in natural language in a textbox. It was then reformatted as a Python docstring, joined to the function signature, and used to query the Code LLM.[6] The generated code was then displayed to the student, along with the result of running the test cases that the student had been shown. Students were free to reattempt the same task repeatedly or to move on to the next task at any time.

After the main study, participants completed a short survey with questions about their experiences in the experiment and demographic questions. We adapted validated instruments from previous work [4, 8, 29, 30] to measure perceptions of the AI tool, mental workload of the task, and participant math anxiety. We also asked questions related to AI ethics and participant perceptions of AI. Finally, there were three open-ended questions that asked participants to reflect on issues or problems they ran into, how they thought Charlie worked, and any other comments on Charlie.

4.2 Evaluation Metrics

4.2.1 Quantitative. We measure participant success on the model-prompting task in two ways. First, we look at their *eventual successes*: did they ever solve the problem during the study? A success occurs when the model generates code that passes all tests in the test suite.

Second, we look at the *reliability* of their prompts in general. Code LLMs are stochastic: they may generate different programs for the same input. A reliable prompt is one that works when it is resubmitted to the model multiple times. We measure this by rerunning each prompt 20 times, and counting how often the generated code is correct. A highly reliable prompt will lead the model to generate a correct solution each time it is queried, for a reliability score of 1 (20/20).[7] We use the open-source StarCoder model [52]

[6] Nguyen et al. [60]'s Figure 4 provides a diagram of this process

to estimate reliability rather than Codex, both for reasons of cost and to aid comparison with the results reported in Nguyen et al. [60].

4.2.2 Qualitative. Participants answered three open-ended questions as part of the post-task survey, the results of which we present below. Both authors were involved in the analysis of the participant responses and employed a lightweight open coding approach. Complete consensus was achievable, given the relatively concise nature of the 201 total data points. The codebook[8] was developed to describe trends in this work and does not provide a general theory.

It is important to note that the same authors also conducted the qualitative analysis for Nguyen et al. [60]. We attempted to mitigate the effects of this previous experience by developing a novel codebook from an inductive analysis process and refraining from actively referring to the Nguyen et al. [60] codebook during analysis.

The coding process was as follows. Each researcher independently developed a series of codes for the participant responses to each question. They then met to discuss the codes, investigate themes amongst the codes, and agree on an intermediate set. They re-coded the data independently using the revised codes. Another session was held to finalize the set of codes and reach consensus on their application to all data points.

Quotes presented below are direct selections from student responses, other than where noted inline. We have anonymized responses that might have disclosed personal identifiable information.

5 RESULTS

Our study investigates the feasability of the natural-language-to-code task for users with no programming experience as a proxy for the possible challenges Code Adapters and Runners may face. We explore quantitative measures of success, discuss participant perceptions of the task, and compare our results with findings from Nguyen et al. [60], which applied the same methodology to study beginning programmers after a single CS1 course.

5.1 Basic Findings

In this section, we present basic findings from our study, addressing the success of non-programmers and broad trends from our qualitative analysis of the open response data related to overall task experience.

5.1.1 Were Non-programmers Successful? Our experiment asked non-programmers to solve CS1 programming tasks by describing them in natural language to a Code LLM. Overall, participants found the task difficult and were only moderately successful, solving, on average, 1.4 of 4 total problems. Moreover, the average prompt reliability score was 0.088 for all prompts and 0.42 for successful prompts. This means that even prompts that were successful during the experiment passed less than half of the time when re-querying the model.

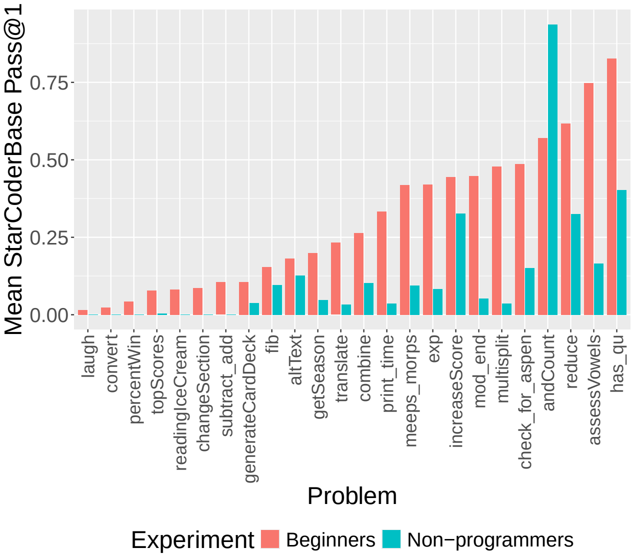


Figure 4: Prompt reliability rates by problem. The graph compares reliability of prompts written by beginners from Nguyen et al. [60] in comparison to non-programmers from the current study

5.1.2 How Do Non-programmers Compare to Beginners? Although many beginning programmers in Nguyen et al. [60] found the task difficult, there is a wide gap between their beginners and our non-programmers. When we compare the reliability of their prompts across problems (Figure 4), the non-programmer prompts are less reliable for all but one problem (discussed in Section 5.3.1).

Table 1 summarizes key quantitative comparisons between the two groups of participants. Note that the participants in Nguyen et al. [60] attempted eight problems; we consider only their first four untimed problems, which we reused in our study. On average, non-programmers solved one fewer problem during the experiment than beginners. This does not reflect lower effort or less engagement, since non-programmers submitted more prompts per problem. This indicates that they were willing to retry problems when they failed, in some cases, many times: one non-programmer submitted 90 prompts, but only solved 2 of the 4 problems; another tried 42 times without ever succeeding.

5.1.3 Are there power users? Non-programmer-written prompts have low prompt reliability scores, even when we consider prompts that succeeded during the experiment. A question that we might ask is whether there are any "power users"– non-programmers with prior experience prompting LLMs who are able to transfer these skills to Code LLMs. In Nguyen et al. [60], the best participants had prompt reliability rates of around 80% (5 participants, solving all or all but 1 problem). In this study, the highest participant prompt reliability rate was 29%; this participant solved only half of the

[7] When benchmarking models, it is common to calculate more lenient measures, such as pass@10 (*if the model is given 10 attempts, does it ever generate a correct solution*?) or even pass@100 (*if the model is given 100 attempts, does it ever generate a correct solution?*). The reliability scores that we report are *pass@1* [15], which has become the standard metric used in the evaluation of Codex [15], GPT-4 [64], Code Llama [6], and others [26, 52, 62]. Following standard practice, we compute this over 20 samples to ensure robustness to the stochasticity inherent to the Code LLM generation process [52].

[8] Available at https://doi.org/10.17605/OSF.IO/MXH35

	Beginners	Non-programmers
Mean Eventual Success Rates	2.6	1.4
Mean Prompt Reliability, All Prompts	0.26	0.088
Mean Prompt Reliability, Successful Prompts	0.62	0.42
Mean Submissions Per Problem	2.75	3.5

Table 1: Quantitative comparisons of prompt performance across two metrics from Nguyen et al. [60]'s beginners and the non-programmers from the current study

Category	N
Wording Matters	28
Charlie Doesn't Understand Me	22
Trouble Writing a Description	22
Couldn't Make Progress	16
We Just Didn't Work Well Together	12
Trouble Understanding the Problem	8
Error Messages	6
Hard To Understand The Output	5
Randomness	3
I am Not a STEM Person	3

Table 2: Codes emerging from responses to *What kinds of problems or issues did you run into working with Charlie?*

problems.[9] Thus, although all participants reported that they had heard of GPT-3 or ChatGPT (Appendix C.2), there is no evidence that our non-programmers were able to transfer prior experience with other LLMs into successful Code LLM prompting strategies.

5.1.4 Communication Gone Awry. Our post-task survey asked participants to reflect on any problems or issues that they ran into when working with Charlie (Table 2). A key trend throughout was a general sense of communication failure. Our participants were less likely to attribute blame to the model than those in Nguyen et al. [60], 19% of whom complained that Charlie rejected valid prompts (n = 23). Instead, our participant's responses reflect an overall feeling of things going wrong without a specific sense of what could be improved. We identified 12 out of 67 non-programmers as reporting "We Just Didn't Work Well Together" (Table 2), or that they were unable to work effectively with the Code LLM and unsure who to blame. For instance, *silverHazelnut* expressed a clear sense that things were going wrong without a clear sense of why: "[...] despite trying multiple variations of instructions, I would still receive incorrect outputs. Maybe my descriptions were not long enough or descriptive enough to get him to do what he needed to do."

5.1.5 Hard to Strategize. A second trend was a sense of getting stuck: after trying various approaches to writing prompts and still not succeeding, participants ran out of strategies to use. They mentioned this throughout their responses (Table 2, Couldn't Make Progress & Charlie Doesn't Understand Me and Table 4, Negative About Task). For instance, *coralAster* commented, "Sometimes he just couldn't understand what I was meaning and would make the same mistakes over and over again." A more emotive description of the same feeling was provided by *pinkIndigo*: "It was definitely frustrating to feel like after putting a lot of thought into this, it didn't work most of the time. It felt like I was trying to explain something to a little kid, but even worse, because I did not know how to relate to the technology in a comprehensible way."

Unlike the beginning programmers in Nguyen et al. [60], who developed a variety of strategies and even, in some cases, sought to systematically test the model's capabilities, non-programmers reported running out of ideas about what to try next. *redHazelnut* noted, "Even though I located the patterns and tried to convey it in understandable terms, for some problems it seemed like nothing would get through." This is a participant who claims to understand the task, but cannot communicate it effectively to the model. Similarly, *yellowIndigo* reported, "I kept trying to think of different ways for Charlie to have the correct output, but after doing it a few times I would get frustrated. This made me not want to help Charlie at all." Overall, our findings suggest that non-programmers may lack the resources to develop effective prompting strategies on their own or even to make progress on the task.

5.2 The Impact of Previous CS Education

A key question driving our research is how to prepare non-expert workers of the future to use Code LLMs. By comparing the struggles of non-programmers to those of beginning students, we can measure the contribution that CS1 makes towards facilitating Code LLM use. How is the task easier for participants who have taken CS1? Are there challenges that still remain? Below we highlight findings related to the impact of previous CS education on student prompting.

5.2.1 Explaining in English. A key challenge highlighted by non-programmers was expressing the task in natural language (Table 2, Trouble Writing a Description). *silverHazelnut* commented, "It was difficult to articulate what I wanted the output to be." Similarly, *plumLupine* wrote, "I felt like I knew what the answers were, but I didn't know how to write them out in the description." These responses show how non-programmers struggle to map problem understanding to natural language.

Beginning students also discussed grappling with this aspect of the task in Nguyen et al. [60]. This suggests that describing technical concepts in natural language is a key skill, but one that takes some time to develop. Although CS1 gives students some practice with expressing technical concepts in natural language, it is an ongoing learning process for beginner programmers, who require more than one traditional computing course to fully develop this skill.

[9] 3 participants solved all four problems, but their prompt reliability scores were very low (0.03-0.15), suggesting they were simply lucky in the moment.

5.2.2 Technical Terminology. Although communicating their intent is a challenge for both non-programmers and beginners, beginners have a key advantage: they are familiar with programming terminology. One of the biggest gaps we observed between our participants and the students in Nguyen et al. [60] was their knowledge of Python vocabulary. Many of the beginning students in Nguyen et al. [60] developed a strategy of using Python keywords in their prompts. In contrast, this strategy was not available to our non-programmers.

In fact, 42% of students brought up an issue related to wording their prompts in the post-task survey (Table 2, Wording Matters). For instance, *bisqueIronweed* wrote, "There seems to be a formula or computer language that I needed to know in order to fluently communicate with Charlie" and *tomatoRedbud* mentioned they "[...] didn't know how to dictate parts of a sequence [...]" Another participant pointed out how this acts as a barrier to non-experts, writing, "If [...] one's ability to program effectively using this tool relies on knowing the keywords, this is pretty inaccessible." (*pinkIndigo*).

5.2.3 Challenges with Vagueness and Specificity. Our participants struggled with the model's inability to handle vague or underspecified descriptions. This echoes themes from work on beginning students [36, 60], including Babe et al. [5]'s finding that to Code LLMs, beginner-written prompts are often underspecified, leading them to generate multiple semantically-distinct programs from a single prompt.

Many non-programmers were critical of this aspect of the model in their post-task survey responses (Table 2, Charlie Doesn't Understand Me and Table 4, Negative About Task). For instance, *orangeStrawberry* wrote that "Charlie required explicit and clear instructions, failing to make small jumps/assumptions", while *yellowIndigo* commented: "I think it would be better if Charlie could take in information that isn't super clear. For example, I think my commands were very reasonable and I even kept changing my prompt to be more clear but it still wasn't working."

Non-programmers pinpointed a specific aspect of technical language overlooked by beginners: the fact that colloquial synonyms cannot be interchanged with technical words. For instance, though "group" and "set" might be used synonymously in ordinary speech, they have specific technical meanings when used to talk about code. Non-programmers were surprised that the model did not understand substitutions of synonyms. For instance, *siennaRedbud* wrote that Charlie "should be able to take into account multiple phrases in order to be able to guess at what you want [...] if I was trying to tell it to change the order I could put change/mirror/flip/order." Similarly, *crimsonIndigo* commented that Charlie "was unable to respond to the differences in language I was using, even though they were synonyms of words [Charlie] recognized."

This is an aspect of technical communication that is rarely discussed explicitly in CS1, but that is certainly practiced throughout the course. Nguyen et al. [60] do not highlight any issues or confusion around this point in their analysis, reflecting how by the time students have completed CS1, the rigid meaning of technical terms is second-nature to them.

5.2.4 Understanding Model Output. Working with Code LLMs is an iterative process: once a prompt is submitted, the user assesses the generated code and, if necessary, modifies their prompt to resubmit. In our study, as in Nguyen et al. [60], the generated code was automatically tested and displayed to students along with the output of the test cases. Beginners in Nguyen et al. [60]'s study used the generated code to figure out how to edit their prompts: around a quarter (n = 29) looked at the generated code before the test cases. Some identified specific issues with generated code, such as its use of unfamiliar Python features (n = 5), which suggests that they were reading the code line-by-line.

By contrast, our non-programmers rarely mentioned the generated code at all; only 5 participants expressed issues with understanding the model output (Table 2, Hard to Understand The Output). This likely reflects a lack of engagement with the code, rather than a lack of issues. Non-programmers who mention the output report struggling to understand it at all, rather than the specific issues identified by beginners [60]. For instance, *crimsonPepperbush* wrote, "Everything I seemed to put into the textbox came out incorrectly. I was just having a hard time understanding the functions and also the answers that [C]harlie produced."

Some participants also discussed struggling with the error messages that were displayed when the generated code raised an error at runtime (Error Messages, n = 6). For instance, *pinkHolly* commented, "Sometimes I hit a wall into something I did wrong and would copy and paste the original description text to try again and see what did. But then it would say run failure and I had no idea why. By the end, the poor cow was confused and so was I." Without being able to read code or understand error messages, non-programmers lacked useful feedback on how to improve their prompting strategies: *pinkLupine*, for instance, "... wish[ed] Charlie was able to describe why the code didn't produce the desired output so [they] could adjust the prompt." These difficulties contributed to the general sense of feeling stuck discussed in Section 5.1.5: when their prompts failed, non-programmers had little information that was useful to them for editing.

5.3 How Do Non-programmers Word Prompts?

Above we consider how non-programmers related to their prompts; here we explore the prompts themselves. We present two cases studies of the natural-language-to-code task: `andCount`, a problem that non-programmers mostly succeeded at, and `readingIceCream`, a problem that was difficult for them.

5.3.1 Prompting Case Study 1: `andCount`*.* The `andCount` task is an outlier: out of all 24 problems, it is the only one where the non-programmers' prompts outperformed the prompts of Nguyen et al. [60]'s beginning students. The reliability of non-programmer-written prompts for this problem was strikingly high: our non-programmer prompts had an average reliability score of 0.94, compared to 0.58 in Nguyen et al. [60] (Figure 4).

`andCount` is a function that takes in a list of four Python strings, containing four characters that are either '-' or '&', and returns the total number of '&' characters in all strings. As an example, `andCount([‘--&-’,‘----’,‘-&--’,‘---&’,])` should return 3. In total, 11 non-programmers attempted this problem, and all but one succeeded on their first attempt.

When we inspect the non-programmer-written descriptions, we find three styles. The most common (9/12) and effective strategy was to describe the problem as simply counting the number of ampersands (e.g., “count the number of & symbols that are given in the input.” – *navyHolly*). One point of variation across all participants was how they described the key character: one used the word “ampersands,” five used & without quotes, and five used quoted strings: either a full example (e.g., contained `‘--&-’`), ‘&’, or “&”. This variation reflects that fact that, without prior programming experience, non-programmers have only the examples and generated code to guide them towards understanding string types and how they are formatted in Python.

The participant who did not succeed at this problem provided the following prompt twice, adding **total** the second time: “state the **total** number of & signs that appear in the bracket.” Notably, this is the only non-programmer who attempted to address the problem’s input type (a Python list).

We posit that non-programmers’ avoidance of describing the data type actually helped them on this problem. In Nguyen et al. [60]’s set of prompts, beginners often make mistakes when they try to be specific about how the data is structured. This is a rare case of when a high-level description with no mention of data types works well, allowing non-programmers to succeed.

5.3.2 Prompting Case Study 2: `readingIceCream`*.* `readingIceCream` exemplifies the other end of the success spectrum. This function takes a list of Python strings and returns the sum of all the numbers in the strings. The input data is complex: each string represents one line of a tab-separated value (tsv) file. For instance, the input `["salty\tfrozen yogurt\t10"]` should produce 10.

The same 11 students attempted this problem as `andCount`. They submitted 63 total prompts – none succeeded during the study and all prompts had a reliability of 0. The maximum number of tests passed was 2 out of 3, achieved by 19% of the prompts (four participants).

We manually inspected the 63 prompts and categorized them into zero or more categories based on three common attributes: (a) mentioning summing up all the numerical values, (b) mentioning the `\t` symbol, a “t”, or the location of the number in the string relative to the tab, and (c) mentioning that the number was at the end of the row or line using only words.

Overall, 76% of prompts (10/11 participants) mentioned adding, totaling, or summing all the numbers. 41% of prompts discussed the location of the number relative to the tab character. Finally, 22% of prompts mentioned that the number appeared at the end of the line. These two categories were mutually exclusive: no one mentioned both the tab and the end of the line, although some mentioned the characters after the tab.

When we consider the task, it seems that a prompt that mentions adding the numbers *and* some information about the location of those numbers in the string should be a sufficient description. We find that 29/63 (46% of) prompts written by 8/11 participants meet this standard. One example is *aquaAster*’s “add values at the end of each line and output the sum.” Notably, none of these 29 prompts resulted in model success. We see `readingIceCream` as reinforcing the findings above: in some cases, students write poor quality prompts due to lack of technical knowledge, while in others, the model fails to understand prompts that do accurately capture all components of the task.

Category	N
Other or Irrelevant Statement	20
Keywords	13
Explicit AI/ML Mention	13
Translation	13
Charlie’s Behavior is Hardcoded	10
Knows About Code	9
Knows About Data	7
Machine Learning is Happening	7
No Idea	4

Table 3: Codes emerging from responses to *How do you think that Charlie works? What is happening behind the scenes?*

5.4 Non-programmer Attitudes

The post-task survey contained scales to quantitatively measure non-programmer perceptions of the task, as well as open-response questions about their experiences. In this section, we discuss how participants felt about the task and about themselves after participating. We provide additional data in Appendix C.

5.4.1 Non-programmer Mental Models. Our post-task survey asked participants to describe how they thought Charlie worked. Since they had no background in CS, we did not expect them to have fully-developed mental models of how the technology worked. However, all participants had heard of either GPT-3 or ChatGPT (Appendix C.2), so they were familiar with the concept of generative AI.

Table 3 summarizes the explanations of the system provided by students. Many students have a keyword-retrieval (19%) or translation (19%) mental model, echoing themes from Nguyen et al. [60]. The same number mentioned AI or machine learning in their responses, though few gave any explanation of how these technologies work. We note that although many beginning students in Nguyen et al. [60] posited a database or dictionary lookup mental model, only one participant response in the Keywords code mentioned the word “database.” It is possible that students develop this incorrect mental model after learning about dictionaries in CS1.

One striking finding is that some students believed Charlie’s responses were hard-coded (Table 3, Charlie’s Behavior is Hardcoded). *bisqueIronweed* wrote, “I think Charlie tries to match our description with the languages/code work that was programmed in Charlie,” and *olivePepperbush* replied that “I think Charlie is probably designed with a set of responses but I don’t really know what goes behind the scenes to make this work.” These participants appear to think that Charlie was not truly generative, but rather, that there was a single correct answer pre-programmed (*pinkRedbud*: “Charlie compares my answers to the correct answers in the machine”), which would be retrieved if their description used the right words (*navyTea*: “I think that the system has been programmed to be able to perform some commands, and when you use those commands correctly, then the system is able to perform the task.”).

5.4.2 How Did Non-programmers Feel About the Task? Throughout our post-task survey, non-programmers commented on the emotional experience of participating in the task. 13% of participants brought up negative emotions in response to our final question

Category	N
Positive About Charlie	26
No	20
Stressed, Frustrated or Confused	9
Positive About Task	6
Negative About Task	6
Negative About Self	5
Comment About Experiment, Not Task	5
Cow Lessens Stress	4
Negative About Charlie	3

Table 4: Codes emerging from responses to *Anything else you'd like to share with us related to Charlie?*

(Table 4, Stressed, Frustrated, or Confused). For instance, *goldSunflower* commented, "The cow was really cute and I wanted to like it but in the end we were both stressed and confused." *tanStrawberry* wrote, "[A]s someone with no coding experience I was very confused at first [...]" Even some participants who ended up liking the task commented on the negative emotions they experienced. For instance, *limeHolly* responded, "It was frustrating definitely, but also kind of fun." So, although there were about equal number of participants who left positive and negative comments about the task, it is clear that the task was, at times, stressful, confusing, or frustrating for many participants.

Several participants (n = 4) explicitly mentioned that the avatar made the task less daunting: *linenTea* noted, "I think the cow aspect made it less intimidating and it was overall pretty fun to do", and *navyRose* wrote, "the cow was pretty cute. I think that lessened some of stress." Although we did not intentionally design the platform with this goal in mind, since our study reused an existing experimental platform, this unexpected finding highlights the impact of UI design decisions for burgeoning programmers [48].

5.4.3 How Did Non-programmers Feel About Themselves? In addition to negative feelings about the task, some non-programmers reported negative feelings about themselves as a result of participating (Table 2, I am Not a STEM Person and Table 4, Negative About Self). For instance, *grayPepperbush* mentioned, "I just wasn't very sure what I was dealing with. I think if I had more of a problem-solving brain it would have been more straight forward." Similarly, *tealPepperbush* commented, "I'm incompetent at coding so someone who knows how to prompt Charlie would meet more success?" Nguyen et al. [60] do not mention any similar instances of negative self-talk.

Compared to the beginning programmers in Nguyen et al. [60], who blamed the model for many miscommunications and pinpointed specific model weaknesses, our non-programmers were more likely to blame themselves when things did not work. For instance, *pinkHolly* commented "Charlie's cute, but I feel incredibly stupid not knowing what was happening." This behavior is particularly striking as participants were specifically recruited to this study *because of* their lack of experience.

One possibility is that the task activated existing anxieties around computational and mathematical skills. Due to institutional demographics, most participants were women, a population among whom math anxiety is more prevalent [29]. As part of the post-task survey, participants rated their level of math anxiety using a 0 to 10 math anxiety scale [4, 29]. We observe an average rating of 5.5 (SD: 3.0)– surprisingly high, given that Hart and Ganley [29] report a mean of 2.44 for all high school graduates, and our participants attend selective colleges.[10] If participants were already math-anxious, taking part in the programming task could have awakened prior feelings of shame or inadequacy.

Whatever the reason for the negative self-feelings reported in our post-task survey, our results suggest that the stakes are high for non-expert interactions with Code LLMs: if things go poorly, non-programmers may take it as evidence that they are not well-suited for programming, rather than evidence that the model is poor or that their prompting could be improved.

6 PREPARING NON-EXPERTS FOR THE CODE LLM FUTURE

The results of our study suggest that non-programmers are not able to effectively work with Code LLMs on their own. We find that non-programmers struggle with numerous aspects of the interaction, and experience stagnation: they run out of approaches to try without discovering an effective strategy (Section 5.1.5). How can we help prepare non-programmers, and other non-experts, for a generative AI future?

Our comparison with Nguyen et al. [60] reveals ways in which a traditional CS1 course does not equip students to effectively use Code LLMs (Section 5.2). For students who study computer science, this is not a problem, because we know that traditionally-trained professional programmers can leverage Code LLMs [7, 11, 53, 59]. But for the future non-expert, our findings suggest that a traditional CS course may not be the best fit in the age of generative AI.

In this section we lay out a vision for training non-experts to effectively work with Code LLMs inspired by both the study conducted here and other related findings. Rather than teaching computer science, or even programming, we envision this training as teaching **code-building**: the ability to solve code-related tasks in a way that moves fluidly between code generation and code adaptation.

We envision a future where a non-expert programmer can level up from a Code Reader or Adaptor to a **Code Builder**: someone who leverages AI to do the tasks of a Code Writer, but with less training. A proficient Code Builder, like a Code Writer, is able to create new programs from scratch, rather than relying on existing code; unlike a Code Writer, a Code Builder accomplishes this via a Code LLM. A successful Code Builder, like a Code Adaptor, will also be able to identify and adapt useful code from a variety of sources.

Below we discuss two current barriers, challenges with responding to the model's output and a lack of technical communication skills, that prevent current non-expert programmers from becoming Code Builders. We begin by discussing possible interventions for these skills gaps, before turning to broader ideas for training future Code Builders and concluding with our concerns about how this future could unfold.

[10] A number of factors could contribute to the elevated math anxiety we observe, including the educational effects of the Covid-19 pandemic, task-related bias, and the high proportion of non-STEM majors in our sample (see Appendix A.4, C.3).

6.1 Bridging the communication gap

Our findings highlight the enduring challenges of technical communication. Just as programmers have grappled for decades with how to effectively communicate in the workplace [3, 10, 51], they are now grappling with how to communicate effectively to LLMs. Our results support a strong emphasis on technical communication for code-building. The low reliability and eventual success rates of prompts show that without training, non-programmers will not use language that leads Code LLMs to produce working code.

Non-programmers identified two distinct challenges in wording their prompts: the challenge of describing their intent in natural language (Trouble Writing a Description, n = 22) and challenge of using the "right" language for the model (Wording Matters, n = 28), in addition to non-specific communication issues (Charlie Doesn't Understand Me, n = 22).

This first challenge emerged both in our study and in Nguyen et al. [60], suggesting that mapping computational intents to natural language descriptions is an ongoing learning process for students beyond CS1. Given the increasing importance of clear technical communication, non-experts should be offered robust opportunities for practicing technical communication skills. The CS education community has long advocated the use of Explain In Plain English problems, which ask students to describe computational concepts in everyday language [17, 54, 76, 79], yet their use has been limited by the challenges of scaling and grading [13, 25]. Nguyen et al. [60] suggest that Code LLMs might provide a way to give feedback for these exercises. This is a useful starting place, but for non-experts, we envision a range of activities that practice technical communication to different audiences: in most workplaces, they will need to be able to communicate not just to LLMs, but also to coworkers across a wide spectrum of computing literacy.

The challenge of programming terminology is specific to non-programmer programmers. The fact that this was not a major theme in Nguyen et al. [60] suggests that current CS1 courses succeed in teaching coding terminology: in most classes, types and terminology are covered in the first week. Educators believe that terminology matters [2, 55] – our work suggests this will remain key in the future. Code-building training should ensure that non-experts have a strong grasp of programming terminology, as well as an understanding of the distinction between colloquial and technical language, which was a point of frustration for our non-programmers.

Although we advocate for the primacy of technical communication in worker skill development, we also believe that model developers should work to address vocabulary issues. There is a significant benefit for all users (both non-experts and experts) if models can be more robust to variations in language choice. Moreover, our case study of `readingIceCream` suggests that there are descriptions that accurately describe the problem to a human, but are not successful model prompts. While some observations may be model-specific, our results indicate fundamental differences between how non-programmers describe a problem versus how experts do.

6.2 Strengthening code understanding

Our study also highlights the challenges that non-programmers face when it comes to evaluating model-generated code. We observe a stark contrast between how non-programmers relate to model output compared to the beginning CS1 students studied by Nguyen et al. [60]. Although both groups discuss struggling to understand model-generated code, the beginning students highlight unfamiliar Python constructs, while our participants report no benefit at all from reading the generated code or error messages. This is not surprising, but it is concerning if non-expert use of Code LLMs is to become widespread: non-experts will not be able to tell whether the generated code is correct, safe, or even relevant to the task.

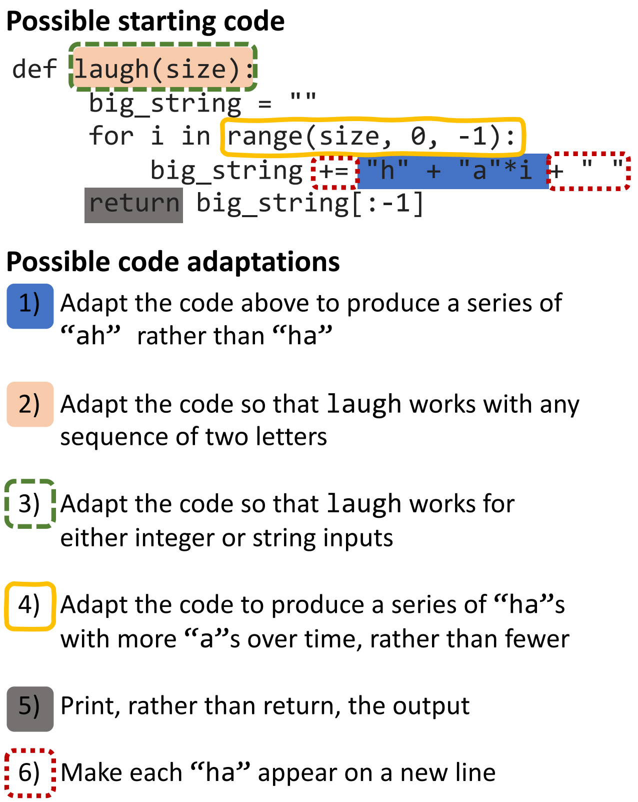
We envision a strong emphasis on *code comprehension* as a primary learning goal in non-expert training. While reading and writing code appear to be related skills [17, 54], most traditional CS1 courses prioritize writing code: assignments typically focus on the task of writing a program, either from scratch or by filling in starter code. However, code comprehension is arguably more important for Code Builders, since they must be able to evaluate model-generated code. Each run of a Code LLM may produce correct output, incorrect output, or a variety of error messages. Code LLM users must be able to judge whether the generated code is correct and relevant to their task. Our study handles this aspect of Code LLM use for our participants by automatically running a pre-written suite of unit tests. However, in real use cases, non-programmers would need to be able to assess code correctness for themselves, either by writing a set of unit tests that specify the intended behavior, or by some other means. Code Builder training should incorporate lightweight methods of teaching about testing and evaluating correctness early on, as well as exercises in code comprehension [50].

Some non-programmers specifically mentioned error messages as a barrier in their survey responses. CS education researchers have focused on how to make error messages more actionable for non-programmers [19, 57]. In fact, recent work has used LLMs to better explain and present error messages [49]. However, Becker et al. [9] survey 50 years of research into possible interventions and conclude that error messages remain a barrier to student understanding. Even though error messages have significant pedagogical importance, many interventions remain challenging to apply in practice. We believe this area of study will endure even as programming otherwise changes because of Code LLMs.

6.3 Code adaptation

We envision Code Builders as adept at adapting code from a variety of sources, both human-written (i.e, from StackOverflow or Github) and model-generated. This is a behavior that has been observed in practice: Kery et al. [38] find that data scientists frequently work by adapting and remixing existing code. We envision adaptation exercises that teach specific programming operations and explore the space of possible programs that can be produced. These exercises would require Code Builders-in-training to recognize "actionable" elements of model-generated code.

As one example, consider the scenarios presented in Figure 5. This figure presents a possible solution for `laugh`, the most challenging problem in both our study and Nguyen et al. [60]. We present 6 possible adaptation exercises, although many more are possible, each targeting a different element of the `laugh` solution. For instance, (4) asks students to consider how `range` works, (5) addresses return types, and (6) considers novel string formatting. We could



Possible starting code

```
def laugh(size):
    big_string = ""
    for i in range(size, 0, -1):
        big_string += "h" + "a"*i + " "
    return big_string[:-1]
```

Possible code adaptations

1) Adapt the code above to produce a series of "ah" rather than "ha"

2) Adapt the code so that laugh works with any sequence of two letters

3) Adapt the code so that laugh works for either integer or string inputs

4) Adapt the code to produce a series of "ha"s with more "a"s over time, rather than fewer

5) Print, rather than return, the output

6) Make each "ha" appear on a new line

Figure 5: Possible ways to transform a code *writing* task into a code *adaptation* task. The possible starting code represents the most difficult problem attempted by participants, `laugh`. The six presented code adaptations suggest possible activities that Code Builders-in-training could undertake. We highlight the Python terms related to the adaptations in the provided possible starting code.

also draw inspiration from existing work in CS education that focuses on identifying errors in code refactoring [63], which exercises similar skills to adaptation: refactoring is essentially adapting one's own code for a new purpose.

6.4 Assessing appropriate use of Code LLMs

Thinking more broadly, effective Code Builders must be able to assess and select among various interaction modes for working with code. One component of this is understanding when it is not effective or appropriate to use Code LLMs.

Current Code LLMs do not articulate a confidence level or know whether their output is incorrect [75]. This makes it vital for users to understand model limitations. Training programmers to trust models implicitly could result in knock-on effects of invalid code in the workplace. Non-expert programmers may encounter increased risk: they may be more easily impressed by model capabilities, since they lack the experience to identify instances of model failure. Although the non-programmers in our study actually rated the model as less knowledgeable and capable than participants in Nguyen et al. [60] (Appendix C), this likely reflects their relative lack of task success, rather than critical evaluation of the generated code. Specific methods to address these issues are worthwhile and currently under study [72]. Work on AI/ML education may also provide perspectives on possible approaches [33, 69].

In addition, non-expert programmers should be taught to consider the broader implications of using Code LLMs. There are important issues around copyright law and generative AI that have yet to be settled [22, 34, 67, 68]. Moreover, in contexts that involve proprietary information, sensitive customer data, or information subject to export control, it may not be appropriate to use Code LLMs at all, since querying the model involves transferring data to the server running the model, or, more commonly, the company providing the Code LLM service. Finally, Code LLMs may be vulnerable to adversarial attacks designed to compromise code security [1].

6.5 Concerns for the future

While our suggestions above are premised on the idea that Code LLMs can help upskill non-expert programmers, we also have concerns about how this future might unfold. At a high level, Code LLMs (and generative AI more broadly) are evolving at a blistering pace. This means researchers are studying Code LLM technology as it is being deployed in real-world environments. This causes a challenging inversion for educators: students have interacted with the technology prior to their ability to adapt curricula [45].

There has been significant discussion [21, 24, 41] about the potential economic knock-on effects of generative AI broadly. Code Builder training should aim to prepare non-expert programmers for such an economic outlook, while keeping in mind the learning expectations of the field. We see this as a very tall ask indeed and one that, without continued discussion, evolution, and critique, could easily fail to do both.

Finally, non-expert programmers comprise a diverse group. An important equity issue surrounding Code LLM deployment is the issue of language barriers. Current Code LLMs offer limited support for languages other than English [52, 77, 78]: although they are typically described as offering natural language-to-code interaction, in fact, they only offer English-to-code. While the non-programmers we study write and read fluently in English, non-expert programmers who use other languages will not benefit from Code LLMs at all unless the technology supports their languages.

6.6 Recommendations

Above we have laid out a vision for how to train non-expert programmers to effectively leverage Code LLM models, emphasizing the following key skills:

- Ability to communicate technical concepts clearly to a range of audiences, including Code LLMs
- Ability to read code and evaluate its quality, predict its behavior, and assess its task relevance
- Ability to modify code from a variety of sources to accomplish new tasks
- Ability to assess when the use of Code LLMs is appropriate

While we believe that worker training could be impactful in this area, our findings also have implications for people who build generative AI for programming.

Recommendations for Code LLM developers. Our work highlights a usability gap between non-programmers and students with just a single semester of computer science education. A key difficulty that our participants faced was the inability to communicate their ideas in a way that the Code LLM understood (Section 5.2). As model developers, we might feel that in these cases, it is the prompt that is to blame: key details may have been omitted, or the description of the input may be inaccurate. Although we feel that non-programmers should be taught how to communicate with Code LLMs, at the same time, Code LLMs should be able to handle more variation in language than they currently do. A model that could handle underspecified prompts would provide a significant boost to non-experts, while also benefiting expert programmers.

In the post-task survey, participants brought up other ways that the model could be more user-friendly. One suggestion was to give feedback on the parts of the prompt that are unclear. *pinkIndigo* noted, "When the response was "none" for the outputs, I was not quite sure what part of my description was wrong, which was frustrating at points. I think it would be really helpful for it to say what part of it needs to be altered." This relates to a key limitation of current LLMs: they are not aware of what they do and do not know. Having models that could identify specific parts of the prompt that are causing uncertainty and ask the user for clarification is one interesting direction for future work.

Finally, a key concern for model developers should be linguistic equity. Although current Code LLMs are impressive, they only benefit users who can communicate in English. Developing multilingual Code LLMs should be a priority.

Recommendations for Code LLM interface developers. A consistent theme in our participant responses was praise of the Charlie avatar (Table 4). Several participants commented that the Charlie avatar made them feel more comfortable performing the task, complementing findings that personified programming may increase motivation [48]. We observed frequent negative self-talk related to programming in our non-programmer responses, a theme that is not reported in Nguyen et al. [60]. This suggests that an engaging, user-friendly design could be useful in confidence-building and retention in learning environments aimed at non-experts, who may arrive with math or programming-related anxieties.

6.7 Limitations

While our sample size and semi-replication form a solid foundation, there are some inherent limitations to our design. The controlled nature of the study design allows for precise data collection regarding prompt-writing and editing, and facilitates comparison between our non-programmers and Nguyen et al. [60]'s beginners. However, the study environment differs in many ways from the real-world environment (see Section 2) of the non-expert programmer, particularly, in the automated testing of generated code and the specific programming tasks.

When studying Code LLMs, the choice of the underlying model can have an impact on factors such as reliability and eventual success. We used Codex to facilitate direct comparison with the data on beginners from Nguyen et al. [60]. However, this has a significant downside: replication of this study (as well as Nguyen et al. [60]) is no longer possible, as Codex is fully deprecated as of January 2024. We see this as an issue for researchers, but also for the study of Code LLMs as a future of work technology more broadly. We strongly encourage the use of open-source Code LLMs [6, 52, 61] to better equalize access to Code LLM technology for researchers and users.

Like prior work, we concentrate on the natural-language-to-code interaction. Our findings may or may not generalize to multi-turn modalities. Given their potential for explaining code as well as generating code, a priority for future work should be investigating how non-experts interact with conversational Code LLMs.

Finally, this study, and our vision for Code Builder training, should be understood through a particular cultural lens. Our empirical evidence is drawn from non-programmers at selective liberal arts colleges, most of whom were non-STEM majors. This population may have more negative impressions of technology than the average non-expert. Non-expert programmers comprise a diverse group, and our study will only generalize so far. We welcome future work which considers these questions in other environments.

7 CONCLUSION

In this paper, we consider how the advent of generative AI tools for programming will impact non-expert programmers. Code LLMs promise to remove barriers to working with code by allowing users to interact via natural language. For many reasons, however, non-experts may not be able benefit from Code LLMs as experts do. Code LLMs therefore pose both a threat and hold out a promise to non-expert programmers: they could be used either to bridge the skills gap between non-experts and experts, or to widen it.

We present qualitative and quantitative results from a study of non-programmer interactions with a Code LLM. We focus on prompt-writing and prompt-editing processes, and identify key struggles that non-programmers face in using Code LLMs effectively: a lack of access to technical vocabulary and an inability to understand code or error messages produced by the model. By comparing our results to Nguyen et al. [60]'s work with beginning programmers, we highlight the ways in which a traditional CS classroom does and does not equip students to use Code LLMs effectively. Moreover, our qualitative results reveal the high stakes involved in non-programmer interactions with Code LLMs: failed interactions can lead non-programmers to believe that they are not well-suited for programming or problem-solving.

Building upon our experimental findings, we proposed key skills that non-expert programmers need in order to leverage Code LLMs in Section 6. We describe a vision for training non-expert programmers to use Code LLMs effectively in order to work towards a generative AI future that bridges the gap between experts and non-experts.

ACKNOWLEDGMENTS

We would like to thank Huyen Kim, Yuneydy Paredes, and Abby Brennan-Jones for their help with study execution. We thank Arjun Guha, Northeastern Research Computing, and the New England Research Cloud for providing computing resources. This work is partially supported by the National Science Foundation (SES-2326174 and SES-2326175). The publication fees for this article were supported by the Wellesley College Library and Technology Services Open Access Fund.

REFERENCES

[1] Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. 2024. TrojanPuzzle: Covertly Poisoning Code-Suggestion Models. arXiv:2301.02344 [cs.CR]

[2] Suad Alaofi and Seán Russell. 2021. Computer Terminology Test for Non-native English Speaking CS1 Students. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 1304–1304.

[3] Maryam Arab, Thomas D. LaToza, Jenny Liang, and Amy J. Ko. 2022. An Exploratory Study of Sharing Strategic Programming Knowledge. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (, New Orleans, LA, USA,) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 66, 15 pages. https://doi.org/10.1145/3491102.3502070

[4] Mark H. Ashcraft. 2002. Math Anxiety: Personal, Educational, and Cognitive Consequences. *Current Directions in Psychological Science* 11, 5 (2002), 181–185. https://doi.org/10.1111/1467-8721.00196 _eprint: https://doi.org/10.1111/1467-8721.00196.

[5] Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q Feldman, and Carolyn Jane Anderson. 2023. StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code. arXiv:2306.04556 [cs.LG]

[6] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Ellen Tan, Yossef (Yossi) Adi, Jingyu Liu, Tal Remez, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Defossez, Jade Copet, Faisal Azhar, Hugo Touvron, Gabriel Synnaeve, Nicolas Usunier, and Thomas Scialom. 2023. Code Llama: Open Foundation Models for Code.

[7] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 85–111. https://doi.org/10.1145/3586030

[8] Christoph Bartneck, Dana Kulić, Elizabeth Croft, and Susana Zoghbi. 2009. Measurement instruments for the anthropomorphism, animacy, likeability, perceived intelligence, and perceived safety of robots. *International journal of social robotics* 1, 1 (2009), 71–81. Publisher: Springer.

[9] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education* (2019), 177–210.

[10] Andrew Begel and Beth Simon. 2008. Struggles of new college graduates in their first software development job. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 226–230. https://doi.org/10.1145/1352135.1352218

[11] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2022. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (2022), 35–57.

[12] 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*. 10–13.

[13] Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, and Craig Zilles. 2020. A Validated Scoring Rubric for Explain-in-Plain-English Questions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 563–569. https://doi.org/10.1145/3328778.3366879

[14] John Chen, Xi Lu, Michael Rejtig, David Du, Ruth Bagley, Michael S Horn, and Uri J Wilensky. 2024. Learning Agent-based Modeling with LLM Companions: Experiences of Novices and Experts Using ChatGPT & NetLogo Chat. *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)* (2024). https://doi.org/10.1145/3613904.3642377

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

[16] Parmit K Chilana, Rishabh Singh, and Philip J Guo. 2016. Understanding conversational programmers: A perspective from the software industry. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 1462–1472.

[17] Malcolm Corney, Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. 2014. 'explain in plain english' questions revisited: data structures problems. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 591–596. https://doi.org/10.1145/2538862.2538911

[18] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2023. Promptly: Using Prompt Problems to Teach Learners How to Effectively Utilize AI Code Generators. https://doi.org/10.48550/arXiv.2307.16364

[19] 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*. 1–15.

[20] Stefania Druga and Amy J Ko. 2021. How do children's perceptions of machine intelligence change when training and coding smart programs?. In *Interaction Design and Children*. 49–61.

[21] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. 2023. Gpts are gpts: An early look at the labor market impact potential of large language models. *arXiv preprint arXiv:2303.10130* (2023).

[22] A. Feder Cooper, Katherine Lee, James Grimmelmann, Daphne Ippolito, Christopher Callison-Burch, Christopher A. Choquette-Choo, Niloofar Mireshghallah, Miles Brundage, David Mimno, Madiha Zahrah Choksi, Jack M. Balkin, Nicholas Carlini, Christopher De Sa, Jonathan Frankle, Deep Ganguli, Bryant Gipson, Andres Guadamuz, Swee Leng Harris, Abigail Z. Jacobs, Elizabeth Joh, Gautam Kamath, Mark Lemley, Cass Matthews, Christine McLeavey, Corynne McSherry, Milad Nasr, Paul Ohm, Adam Roberts, Tom Rubin, Pamela Samuelson, Ludwig Schubert, Kristen Vaccaro, Luis Villa, Felix Wu, and Elana Zeide. 2023. Report of the 1st Workshop on Generative AI and Law. *arXiv e-prints*, Article arXiv:2311.06477 (Nov. 2023), arXiv:2311.06477 pages. https://doi.org/10.48550/arXiv.2311.06477

[23] Molly Q Feldman, Ji Yong Cho, Monica Ong, Sumit Gulwani, Zoran Popović, and Erik Andersen. 2018. Automatic diagnosis of students' misconceptions in k-8 mathematics. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[24] World Economic Forum. 2023. Future of Jobs Report. https://www3.weforum.org/docs/WEF_Future_of_Jobs_2023.pdf

[25] 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* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1163–1169. https://doi.org/10.1145/3408877.3432539

[26] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[27] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*. 653–663.

[28] Philip J. Guo. 2023. Six Opportunities for Scientists and Engineers to Learn Programming Using AI Tools Such as ChatGPT. *Computing in Science and Engineering* 25, 3 (May 2023), 73–78. https://doi.org/10.1109/MCSE.2023.3308476

[29] Sara A. Hart and Colleen M. Ganley. 2019. The Nature of Math Anxiety in Adults: Prevalence and Correlates. *Journal of numerical cognition* 5, 2 (2019), 122–139. https://doi.org/10.5964/jnc.v5i2.195 Place: Germany.

[30] Sandra G. Hart and Lowell E. Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In *Advances in Psychology*, Peter A. Hancock and Najmedin Meshkati (Eds.). Human Mental Workload, Vol. 52. North-Holland, 139–183. https://doi.org/10.1016/S0166-4115(08)62386-9

[31] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 89–98.

[32] Dhanya Jayagopal, Justin Lubin, and Sarah E. Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages. https://doi.org/10.1145/3526113.3545659

[33] Juho Kahila, Henriikka Vartiainen, Matti Tedre, Eetu Arkko, Anssi Lin, Nicolas Pope, Ilkka Jormanainen, and Teemu Valtonen. 2024. Pedagogical framework for cultivating children's data agency and creative abilities in the age of AI. *Informatics in Education* (2024).

[34] Antonia Karamolegkou, Jiaang Li, Li Zhou, and Anders Søgaard. 2023. Copyright Violations and Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 7403–7412. https://doi.org/10.18653/v1/2023.emnlp-main.458

[35] 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*. ACM, Hamburg

Germany, 1–23. https://doi.org/10.1145/3544548.3580919

[36] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2024. 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* (, Koli, Finland,) *(Koli Calling '23)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. https://doi.org/10.1145/3631802.3631806

[37] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Z. Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. arXiv:2401.11314 [cs.HC]

[38] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3025453.3025626

[39] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3173574.3173748

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

[41] Karin Kimbrough and Mar Carpanelli. 2023. Preparing the Workforce for Generative AI. https://economicgraph.linkedin.com/content/dam/me/economicgraph/en-us/PDF/preparing-the-workforce-for-generative-ai.pdf

[42] Donald Ervin Knuth. 1984. Literate programming. *The computer journal* 27, 2 (1984), 97–111.

[43] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[44] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

[45] Sam Lau and Philip Guo. 2023. From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (, Chicago, IL, USA,) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 106–121. https://doi.org/10.1145/3568813.3600138

[46] Sam Lau, Sruti Srinivasa Srinivasa Ragavan, Ken Milne, Titus Barik, and Advait Sarkar. 2021. TweakIt: Supporting End-User Programmers Who Transmogrify Code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 311, 12 pages. https://doi.org/10.1145/3411764.3445265

[47] Tessa Lau. 2009. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine* 30, 4 (2009), 65–65.

[48] Michael J Lee and Amy J Ko. 2011. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*. 109–116.

[49] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using large language models to enhance programming error messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 563–569.

[50] Colleen M. Lewis. 2023. Examples of Unsuccessful Use of Code Comprehension Strategies: A Resource for Developing Code Comprehension Pedagogy. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (, Chicago, IL, USA,) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3568813.3600116

[51] Paul Luo Li, Amy J. Ko, and Jiamin Zhu. 2015. What Makes a Great Software Engineer?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 700–710. https://doi.org/10.1109/ICSE.2015.335

[52] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and others. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[53] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. Understanding the Usability of AI Programming Assistants. (2023). https://doi.org/10.48550/arXiv.2303.17125

[54] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth international Workshop on Computing Education Research (ICER '08)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[55] James J Lu and George HL Fletcher. 2009. Thinking about computational thinking. In *Proceedings of the 40th ACM technical symposium on Computer science education*. 260–264.

[56] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021). https://doi.org/10.48550/ARXIV.2102.04664

[57] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. 499–504. https://doi.org/10.1145/1953163.1953308

[58] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 291–301.

[59] Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, and Nachiappan Nagappan. 2023. CodeCompose: A Large-Scale Industrial Deployment of AI-assisted Code Authoring. http://arxiv.org/abs/2305.12050 arXiv:2305.12050 [cs].

[60] Syndey 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)*. https://doi.org/10.1145/3613904.3642706

[61] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. arXiv:2305.02309 [cs.LG]

[62] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[63] Eduardo Oliveira, Hieke Keuning, and Johan Jeuring. 2023. Student Code Refactoring Misconceptions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 19–25. https://doi.org/10.1145/3587102.3588840

[64] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs]

[65] James Prather, Paul Denny, Juho Leinonen, David H. Smith IV au2, 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:2401.10759 [cs.HC]

[66] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Transactions on Computer-Human Interaction* (Aug. 2023), 3617367. https://doi.org/10.1145/3617367

[67] Md Rafiqul Islam Rabin, Aftab Hussain, Mohammad Amin Alipour, and Vincent J. Hellendoorn. 2023. Memorization and generalization in neural code intelligence models. *Information and Software Technology* 153 (jan 2023), 107066. https://doi.org/10.1016/j.infsof.2022.107066

[68] Noorjahan Rahman and Eduardo Santacana. 2023. Beyond Fair Use: Legal Risk Evaluation for Training LLMs on Copyrighted Text. In *Proceedings of the First Workshop on Generative AI and Law*. Honolulu.

[69] Saman Rizvi, Jane Waite, and Sue Sentance. 2023. Artificial Intelligence teaching and learning in K-12 from 2019 to 2022: A systematic literature review. *Computers and Education: Artificial Intelligence* 4 (2023), 100145. https://doi.org/10.1016/j.caeai.2023.100145

[70] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces* (Sydney, NSW, Australia) *(IUI '23)*. Association for Computing Machinery, New York, NY, USA, 491–514. https://doi.org/10.1145/3581641.3584037

[71] Nikhil Singh, Guillermo Bernal, Daria Savchenko, and Elena L. Glassman. 2022. Where to Hide a Stolen Elephant: Leaps in Creative Writing with Multimodal Machine Intelligence. *ACM Transactions on Computer-Human Interaction* (Feb. 2022). https://doi.org/10.1145/3511599

[72] Jiao Sun, Q Vera Liao, Michael Muller, Mayank Agarwal, Stephanie Houde, Kartik Talamadupula, and Justin D Weisz. 2022. Investigating explainability of generative AI for code through scenario-based design. In *27th International Conference on Intelligent User Interfaces*. 212–228.

[73] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3491101.3519665 event-place: New Orleans, LA, USA.

[74] Kurt VanLehn. 1989. Student modeling. *Proceedings of the Air Force Forum for Intelligent Tutoring Systems* (1989), 49–63.

[75] Helena Vasconcelos, Gagan Bansal, Adam Fourney, Q. Vera Liao, and Jennifer Wortman Vaughan. 2023. Generation Probabilities Are Not Enough: Exploring the Effectiveness of Uncertainty Highlighting in AI-Powered Code Completions. arXiv:2302.07248 [cs.HC]

[76] Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop* (Berkeley, CA, USA) *(ICER '09)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1584322.1584336

[77] Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. 2022. MCoNaLa: A Benchmark for Code Generation from Multiple Natural Languages. https://doi.org/10.48550/ARXIV.2203.08388

[78] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-Based Evaluation for Open-Domain Code Generation. In *Conference on Empirical Methods in Natural Language Processing*. https://api.semanticscholar.org/CorpusID:254877069

[79] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine 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 - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 243–252.

[80] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 29 (mar 2022), 47 pages. https://doi.org/10.1145/3487569

[81] Shuyin Zhao. 2023. GitHub Copilot Now Has a Better AI Model and New Capabilities. https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/

[82] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 21–29.

A ADDITIONAL METHODOLOGICAL DETAILS

A.1 Respect for Participants

Our study was conducted under oversight from our colleges' Institutional Review Boards. We obtained informed consent from each participant at the beginning of the experiment using a Qualtrics form. We compensated participants with a $30 USD Amazon gift card. The study was advertised as lasting 45 minutes; however, most participants finished earlier. Participant prompts and the generated code are released publicly via the Open Science Framework at https://doi.org/10.17605/OSF.IO/MXH35.

A.2 Qualitative Analysis of Demographic Responses

As discussed in Section 4.2.2, the qualitative analysis of the open-ended response questions in the post-task survey was performed by both authors and makes up the primary qualitative dataset.

We also collected demographic information which contained open-ended response questions by replicating Nguyen et al. [60]'s approach. We present results which include data from two such questions, one which asked about a participant's major and the second which asked about languages spoken in their household as a child, in Apendix A.4 and Appendix C. As these questions were exact copies of the questions from Nguyen et al. [60], and the goal of aggregating responses was to present additional participant-wide trends, a single author employed the same process as used in Nguyen et al. [60] to categorize Language and Major data.

A.3 Screening Participants for Prior Programming Experience

We screened each participant at the beginning of the session to make sure that they were eligible for the study. We asked questions about their prior programming experience. Some examples of questions asked include:

- Can you tell me a bit about any programming experience you have had in the past?
- Have you built a website?
- Have you ever worked with statistical software like R or Stata?
- Did you program in any classes in high school, or instance, using Scratch in a Math course?
- Have you ever done an internship or after-school club that involved coding?

A.4 Participant Demographics

Table 5 displays aggregated demographic information about participants collected as part of the post-task survey.

B ADDITIONAL FINDINGS FROM STUDY OF NON-PROGRAMMERS

B.1 Eventual Success Rates

Figure 6 shows the eventual success rates by individual problem for our participants compared to the participants from Nguyen et al. [60]. For the most part, these trends are similar to those in prompt reliability across problems, as discussed in Section 5.1.

Category	Response	N
International Student Status	International	4
	Domestic	63
First Generation College Student Status	First Generation	7
	Not First Generation	60
High School Attended	Public	47
	Private	18
	Other	2
Languages Spoken in Household As Child	Monolingual in English	39
	Monolingual Not in English	10
	Multilingual	18
Major Division	Natural Science	18
	Social Science	29
	Humanities	25

Table 5: Participant Demographics

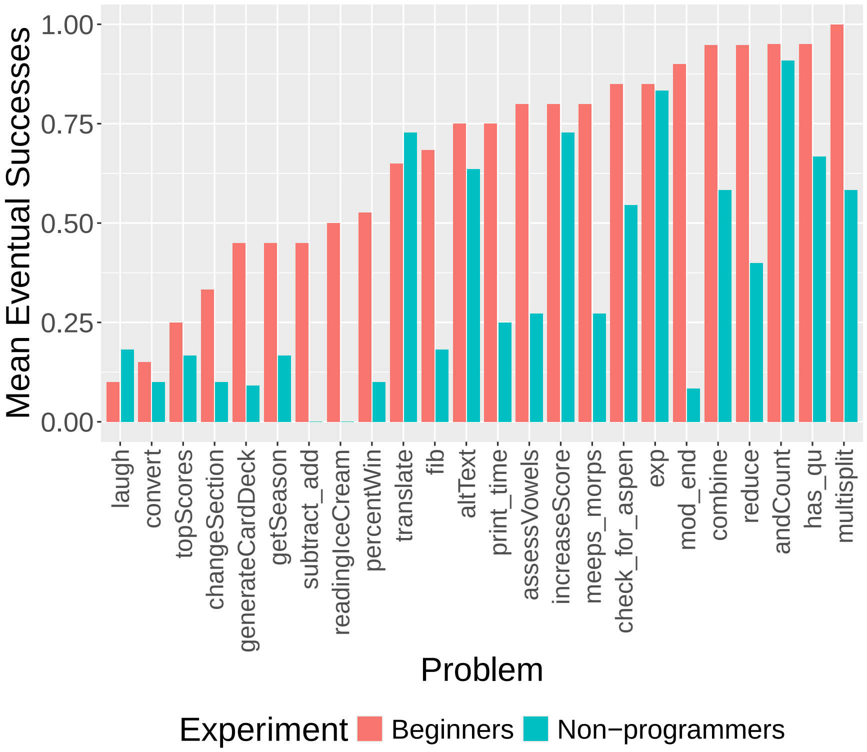


Figure 6: Mean eventual success rates for student-written prompts from this study's non-programmers and Nguyen et al. [60]'s beginners

Figure 8 shows the eventual success rates by problem category for our participants compared to the participants from Nguyen et al. [60]. Somewhat surprisingly, we find that non-programmers do best in the loop category, and worst in the string category, the opposite of Nguyen et al. [60]'s beginning students. We hypothesize that non-programmers struggled with the right way to refer to data types like lists and strings, but were able to avoid problem decomposition in many loop problems, while beginners attempt to describe these tasks step-by-step, but were unable to do so effectively.

C ADDITIONAL FINDINGS FROM POST-TASK SURVEY

In this section, we report additional findings from the post-task survey. We measure the statistical reliability of measure differences using Student's t-tests, with a significance level of α=0.05.

AI Perception Scale	Current	Nguyen et al. [60]	p-value
"Ignorant - Knowledgeable"	3.21	3.68	**0.003**
"Machinelike - Humanlike"	1.79	2.39	**<0.0001**
"Responding rigidly - Responding elegantly"	2.70	3.13	**0.009**
"Unfriendly - Friendly"	3.91	4.20	0.08
"Incompetent - Competent"	3.13	3.58	**0.002**

Table 6: Non-programmer and beginning student ratings of the model on scales from Bartneck et al. [8] (scale ranges from 1-5)

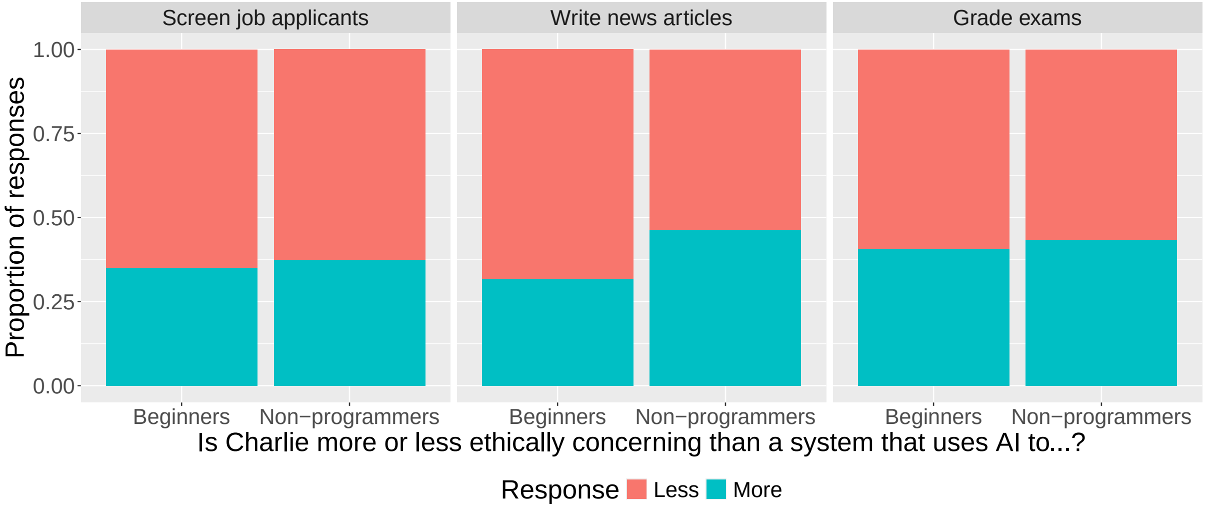


Figure 7: Ethical comparisons between AI applications, non-programmers compared to beginners from Nguyen et al. [60]

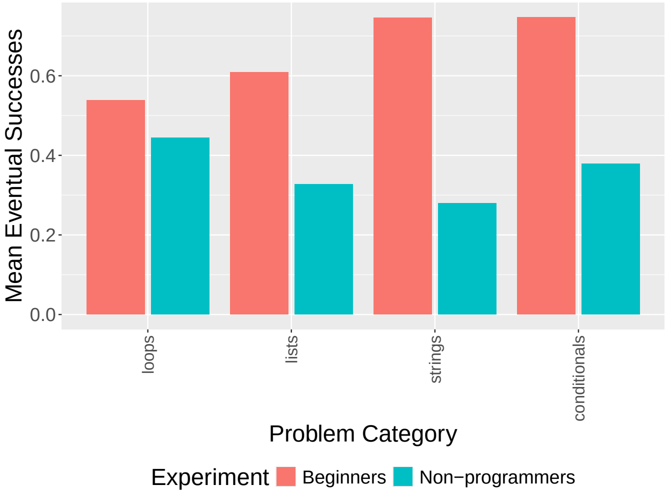


Figure 8: Eventual success rates by problem category (Loops, Lists, Strings, Conditionals) from this study's non-programmers and Nguyen et al. [60]'s beginners

C.1 AI Perception Ratings

As part of the post-task survey, participants rated Charlie's qualities using Bartneck et al. [8]'s AI Perception scale.[11] Overall, participants gave the model fairly high ratings on friendliness, competence, and knowledgeability (Table 6). Although the broad trend is similar to what was found in Nguyen et al. [60], non-programmers in our study gave significantly lower ratings for competence and knowledgeability compared to their beginning students. We posit that non-programmers perceived the model as less capable because of how often communication failed. Non-programmers were also significantly more likely to rate the model as machine-like; this aligns with the remarks non-programmers made about the model's inflexibility with respect to wording.

The post-task survey asked participants for any other thoughts about Charlie. Somewhat surprisingly, most comments related to the Charlie avatar, though a smaller number addressed the task or experimental design. A large number of participants expressed appreciation of the Charlie persona or avatar. An illustrative example is *thistleHazelnut*'s "I think the cow is very cute!" However, one participant did express discomfort with the anthropomorphism of AI.

[11] Similar adaptation for an educational use case was done by Druga and Ko [20].

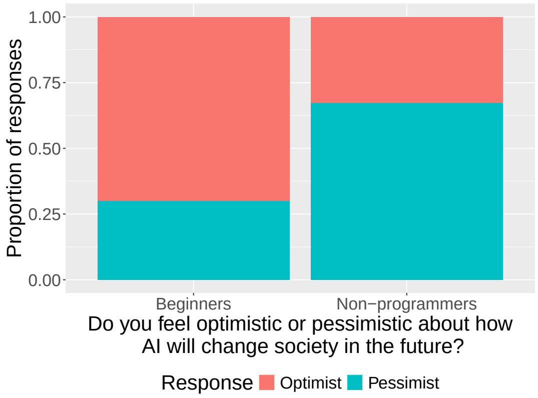


Figure 9: Optimism or pessimism about AI's impact, non-programmers compared to beginners from Nguyen et al. [60]

C.2 AI Familiarity

Table 7 shows how many participants reported having heard of a popular LLM or Code LLM before the study.

Codex/Copilot	GPT-3	ChatGPT	N
X	X	✓	47
X	✓	✓	6
✓	X	✓	9
✓	✓	✓	3
X	✓	X	2

Table 7: Responses to *Have you heard of Codex, Copilot, GPT-3, or ChatGPT?*

C.3 Math Anxiety By Major

Table 8 shows participant ratings on the Math Anxiety question by major [4, 29]. Students majoring in a Natural Science, on average, rated themselves as less math-anxious than students majoring in a Humanities or Social Science field.

Major Division	Mean Math Anxiety Rating
Humanities	6.36
Social Science	6
Natural Science	3.89

Table 8: Average Math Anxiety Ratings by Major Division (scale ranges from 0-10)

C.4 Non-programmer Outlooks on AI

Our post-task survey also explored student attitudes towards AI more broadly. When asked whether they felt more optimistic or pessimistic about how AI will impact society, about two-thirds of our non-programmers identified as pessimists, an inversion of the findings from Nguyen et al. [60] (Figure 9).

We also asked students to compare the ethicality of the Code LLM they used to three other uses of AI in the workplace. Most participants indicated that the Code LLM was less ethically concerning than using AI to screen job applicants, to write news articles, and to grade exams (Figure 7). However, compared to Nguyen et al. [60], more participants felt that writing news articles with AI was less ethically concerning. This went against our expectations, since the non-programmers were drawn heavily from non-STEM majors, and may be more at risk of losing employment opportunities from AI writing systems. However, these participants may envision using AI writing systems to facilitate, rather than automate, their future work.