How Beginning Programmers and Code LLMs (Mis)read Each Other

Sydney Nguyen
Wellesley College
USA

Hannah McLean Babe
Oberlin College
USA

Yangtian Zi
Northeastern University
USA

Arjun Guha
Northeastern University and Roblox
USA
a.guha@northeastern.edu

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

Molly Q Feldman
Oberlin College
USA
mfeldman@oberlin.edu

ABSTRACT

Generative AI models, specifically large language models (LLMs), have made strides towards the long-standing goal of text-to-code generation. This progress has invited numerous studies of user interaction. However, less is known about the struggles and strategies of non-experts, for whom each step of the text-to-code problem presents challenges: describing their intent in natural language, evaluating the correctness of generated code, and editing prompts when the generated code is incorrect. This paper presents a large-scale controlled study of how 120 beginning coders across three academic institutions approach writing and editing prompts. A novel experimental design allows us to target specific steps in the text-to-code process and reveals that beginners struggle with writing and editing prompts, even for problems at their skill level and when correctness is automatically determined. Our mixed-methods evaluation provides insight into student processes and perceptions with key implications for non-expert Code LLM use within and outside of education.

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:
Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24), May 11–16, 2024, Honolulu, HI, USA.* ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/3613904.3642706

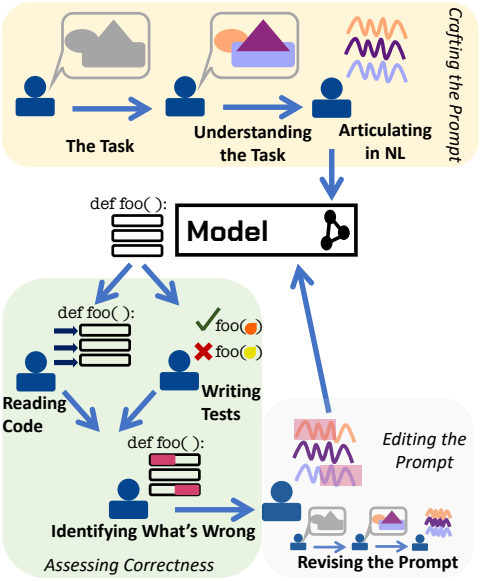


Figure 1: Visualization of the multi-step process of querying a large language mode of code (Code LLM). The user starts with crafting their prompt in natural language (NL). They provide the prompt to the model, which produces code. The user then assesses the correctness of the generated code. If there are errors, they must identify how to resolve them and how to edit the prompt. This continues in an iterative fashion.

1 INTRODUCTION

Computer scientists have been working towards programming in natural language for decades [4, 38, 86], often with the goal of making programming easier for a broader set of users. Recent advances in *generative AI* have brought us nearer to this goal. In programming, along with fields like digital art [79, 81, 83], creative writing [2, 45, 68], and digital music [1, 63], generative AI has reduced the technical skills that users need by allowing them to *prompt* a model with a natural language description of their desired output.

In many fields, experts have started to use generative AI to accelerate their work, including in software engineering, where *large language models of code* (Code LLMs) have enhanced expert programmer productivity [69, 76, 105]. However, to fulfill their potential of democratizing these fields, models must be usable without extensive technical training at each stage of creation: 1) writing

CHI '24, May 11–16, 2024, Honolulu, HI, USA

prompts for the model, 2) evaluating model output for quality, and 3) iteratively refining prompts when generation is unsuccessful.

Programming presents a particularly challenging domain for non-experts. Like art, computer science has evolved an extensive technical vocabulary; since generative models are trained largely on professional code, they may not work as well if users lack this vocabulary. In visual art, music, and creative writing, a user can quickly determine whether they like the generated output even if they are not an expert (embodying the cliché "I don't know anything about art, but I know what I like"). However, this attitude does not extend to programming. It is very challenging for a non-expert to evaluate the quality of a generated program. Even when a user knows enough to determine a generated program is incorrect, they also need to understand it well enough to know what needs to change and how to update their prompt.

In order to use a Code LLM, non-experts must grapple with a multi-step process (Figure 1). First, they must have a clear understanding of what they want the code to do. This may seem trivial, but research on requirements engineering has shown that it can be challenging [75]. Next, the user must clearly articulate the intended behavior of the program in natural language to the model. Once the model generates code, the user must evaluate its correctness by reading it or writing tests. If the code is not correct, they must determine what has gone wrong, and update their prompt accordingly. This requires not only understanding the generated code, but also, understanding the model's generative process. These barriers mirror well-known challenges for non-experts with end-user programming [50] and classical AI systems [53].

There is a growing body of work studying how non-expert programmers use AI-assisted programming systems in naturalistic settings [48, 78]. However, in open-ended tasks, it is difficult to decouple the steps of the code generation process, since they feed each other: if the user fails to identify incorrect code and moves on, their editing process can't be observed. We present results from a carefully-controlled experiment targeting two steps in the code generation process: prompt creation (*How do users describe the intended program in natural language?*) and prompt modification (*How do users modify their prompts when a generated program is incorrect?*).

One challenge in studying how non-experts use Code LLMs is selecting tasks that make sense to them. For example, replicating Barke et al. [6]'s insightful study of experienced programmers would not be appropriate for novices, because the tasks presuppose technical knowledge. Novices have diverse goals, backgrounds, and familiarity with mathematical and computational thinking. Our solution is to target a large population of near-novices with similar experience levels: university students who have completed a single introductory computer science course (CS1). This allows us to select tasks that are conceptually familiar to them.

Our Approach. We ask whether students who have completed CS1 can effectively prompt a Code LLM to solve tasks from their previous course. In order to isolate students' experiences in writing and editing prompts, our experiment presents tasks as input/output pairs and tests the generated code for correctness. This provides in-depth insight into the processes they develop for describing code in natural language and iteratively refining their prompts. We pose three main research questions:

- RQ1: Can students who have completed a CS1 course effectively prompt a Code LLM to generate code for questions from their previous courses?
- RQ2: What is the origin of student challenges with Code LLMs? Do these differ across different groups of students?
- RQ3: What are students' mental models of Code LLMs and how do they effect their interactions?

We find that students struggle significantly with this task, even though we pose problems tailored to their skill level and test code correctness for them. In essence, *beginning programmers and current Code LLMs tend to misread each other*: the Code LLM fails to generate working code based on student descriptions and students have a hard time adapting their descriptions to the model. Our study has concerning implications for democratizing programming: if these students, who already have basic skills in code explanation and understanding, struggle with this simplified task, the full natural language-to-code task—where the user has to determine correctness themselves—must be very challenging indeed for true novices. This finding also has important implications for education. Code LLMs have sparked an intense debate over the future of computing education, including claims that traditional programming training is no longer necessary [65, 100]. By contrast, our findings highlight the continuing importance of teaching students technical communication and code understanding.

Our work differentiates itself from previous work in three key ways: scale, population, and experimental design. First, we study 120 students solving 48 different programming problems. To our knowledge, no previous work has studied user interactions with Code LLMs at this scale. Second, we focus on a near-novice population with fairly uniform levels of experience, allowing us to carefully tailor tasks to their skill level. Finally, we use an experimental paradigm that allows us to isolate the prompt writing and editing aspects of the task.[1]

2 RELATED WORK

Our work focuses on how programmers use LLMs to turn natural language into code. Programming with natural language is a decades old proposition [67] and has led to several ideas about bringing programming closer to how users communicate [70]. For instance, Hindle et al. [39] imagined that future language models could be effective at turning natural language to code, a prediction that has been borne out with Code LLMs.

By exploring beginner interactions with Code LLMs, our study contributes to a growing body of work on how non-experts interact with emerging automated technologies [98], ranging from automated feedback [22, 44, 94] to augmented reality [42, 80]. We situate our study within existing work on user interactions with Code LLMs below.

Experienced programmers and LLMs. We study how beginning programmers interact with a Code LLM, the same foundational technology that powers autocomplete tools such as GitHub Copilot and others [14, 15, 95]. These tools are promoted as productivity-boosting technology for experienced programmers. Recent in-the-wild studies and surveys indicate that these tools are popular with

[1]Data collected as part of this work is publicly available at https://doi.org/10.17605/OSF.IO/V2C4T.

expert programmers, improve their self-perception of productivity, and shift their work from writing code to understanding LLM outputs [10, 60, 69]. In contrast, our study of beginners' interactions with a Code LLM reveals that (1) they have mixed success with writing natural language prompts, (2) and they often struggle to understand LLM-generated code.

Vaithilingam et al. [96] present the earliest academic study of GitHub Copilot with 24 students (undergraduate–PhD) and three tasks. Their main finding is that although participants enjoyed using it, Copilot did not help them code faster or write more correct code. We design our study for less experienced participants. For example, we developed a web interface that is much simpler than a professional IDE. The same study reports that their participants often struggled to validate LLM-generated code, and we avoid this by testing generated code for our participants automatically.

Since Copilot is a general autocomplete tool, one can use it in several ways: to produce code given code, to generate documentation from code, to turn natural language into code, and so on. Grounded Copilot [6] studies experienced programmers and reports that they prefer using it to turn natural language into code [6, Section 4.2.3]. Thus our study design focuses on the natural language to code task, but with beginning programmers.

Non-experts and LLMs. Like us, several researchers have considered the impact of using Code LLMs for the text-to-code task with non-experts, specifically in educational settings. Our work is larger in scale than prior work (120 students from 3 institutions and 48 problems in 8 categories), which allows us to perform statistical analyses that require large sample sizes to be reliable. Moreover, our experiment design allows us to investigate key research questions that prior work has not been able to ask, such as identifying the prompting strategies that beginners use, determining how they modify prompts that do not work, and studying several factors that affect their success.

Prather et al. [78] study 19 students using Copilot for a final project in a CS1 course: building the game Minesweeper. They found that students struggled to use Copilot, even over the course of a week. We reach a similar conclusions with our study, with 48 problems that are much simpler than building a working video game.

Kazemitabaar et al. [48] develop CodingSteps, a web-based Python learning environment that allows users to query Codex. The paper compares 33 participants (10–17 years old) with access to Codex to 36 students programming independently, working on the same set of 45 programming problems over several weeks. Their findings indicate that Code LLMs may benefit student learning outcomes. However, because CodingSteps presents students with expert-written problem descriptions, their results do not shed light on whether beginners can write natural language prompts independently. They report that 32% of student prompts are verbatim copies of the expert-written problem descriptions. In contrast, our study is carefully designed to avoid this problem by showing students input/output examples instead of natural language descriptions. We also investigate the strategies that students use to understand model output and modify their prompts. Kazemitabaar et al. [48] do not address these kinds of questions, partly because their students received feedback from instructors throughout the experiment.

Promptly [19] studies 54 students writing prompts for three CS1 problems. Our substantially larger scale (120 students and 48 problems) allows us to explore research questions beyond what they study, such as the how students change their prompting strategies, and demographic factors that influence success rates. Our paper also presents a detailed analysis of LLM output, such as the kinds of errors that appear in LLM-generated code, and the impact of non-determinism on participants' success.

Lau and Guo [52] interviewed 20 CS1/CS2 instructors in early 2023 about their perceptions of ChatGPT and LLM technologies. They report that instructors hold a diverse set of perspectives: some wanted to "ban it" and others felt urged to integrate these technologies into curricula to prepare students for future jobs that may require using LLM technology. The students in our study echo many of the concerns and desires raised by instructors in Lau and Guo [52].

It is also possible to use language models to assist students learning to program, without having the model write code for the student. For example, Geng et al. [31] use language models to localize type errors in OCaml, but not to correct them. Like our study, this work isolates the interaction mode in which students use Code LLMs; however, we study prompt writing and editing, while they study error detection and explanation.

Alternatives to inline code completion. Copilot and related tools suggest inline code completions, but there are other ways to interact with AI-assisted programming tools. Vaithilingam et al. [96] present new interfaces for Visual Studio that present code changes. Liu et al. [62] build a new interaction model, grounded abstraction matching, which targets spreadsheets and data frames, constraining the generated code to support grounding. These ideas are exciting parallel directions for Code LLM interaction in addition to the natural language prompting approach we study here.

Code LLMs beyond text-to-code. For a beginning programmer, feedback from an expert teacher or teaching assistant can be invaluable. However, access to expert feedback is limited. There is a long line of research that tries to address this shortage by developing systems that generate actionable feedback for students [37, 40, 82, 90, 94]. Phung et al. [77] show that LLMs can help build these systems and generate higher quality feedback than prior rule-based approaches. In contrast to our human experiment, they evaluate on benchmark problems. Moreover, their system is intended to help beginners write code directly, whereas our experiment focuses on prompt writing.

Another body of work focuses on automated program repair [34], which can be used to fix trivial mistakes that frustrate beginners. Traditional automated program repair systems have required significant engineering for each programming language and problem domain. Joshi et al. [47] show that an LLM trained to generate code can be employed to repair simple coding mistakes.

Similarly, Leinonen et al. [55] report that Code LLMs are better at explaining code than beginning students, and Leinonen et al. [56] show that an LLMs explanation of a program error can be better than default error messages. This is further evidence that LLM technology may help students learn to write code directly.

Recent additional efforts include Finnie-Ansley et al. [25], who report that Codex is remarkably good at generating code from

natural language prompts from a CS1 class and several variations of the Rainfall Problem; Dakhel et al. [17], who compare the quality of Codex-generated code to student-written code; and Babe et al. [3], who use student-written prompts to benchmark Code LLMs. Finally, Code LLMs have applications that go beyond natural-language-to-code, and researchers are using them as building blocks for a variety of other tasks [5, 12, 23, 26, 47, 57, 69, 71, 77, 84, 87, 101]. The aforementioned papers present new tools, benchmarks, and studies of LLM capabilities. But, they do not study users' abilities to prompt models, which is the focus of our work.

Using LLMs for non-programming tasks. Researchers are currently exploring a wide variety of applications for LLMs beyond computational tasks. While we do not survey the full range of such work, two recent papers are particularly relevant to our task. Zamfirescu-Pereira et al. [103] study non-experts prompting an LLM to produce recipes. Their participants actively avoided systemic testing, which we address by automating testing. Like them, we find that participants' mental models of LLMs are very different from how they actually work. Singh et al. [89] compare user interactions with a multimedia writing interface with LLM-generated audio, text, and image suggestions. Our post-study interview and survey was inspired by their exploration of participant's perceptions of AI.

3 STUDY DESIGN

Our work explores whether beginning programmers can effectively prompt Code LLMs. We investigate this question through a multi-institutional [24], lab-based study, asking 120 students who completed a CS1 course to describe 8 out of 48 possible problems presented via input/output examples.

In this section, we discuss three major aspects of our study design:

(1) Why do we use a controlled experiment?
(2) How do we successfully present problems to students?
(3) How do we select problems that are appropriate for students?

We discuss the logistics of implementing the study in Section 4.

3.1 Experimental Environment: In the Lab vs. In the Classroom

Studies of student interactions with programming tools can be grouped into three main categories: studies within the context of a course during the term, post-hoc analyses of educational data, or controlled, lab-based experiments. Post-hoc analyses are not currently possible, since there is a lack of available educational Code LLM data. We discuss the decision between a course-based study and lab-based study below.

There are many benefits to real-world studies conducted in a course context, including ease of access to participants and normalized educational background [78]. It is easier to study how technology directly impacts learning by using it alongside instruction [48] or as an evaluative method [44]. At the same time, these studies cannot be as easily controlled: participation may be optional (only around 12% of students chose to participate in Denny et al. [19]); participants may explicitly be learning through the task, making it hard to compare their responses across problems [48]; and in-depth interviews are challenging to conduct.

Lab-based studies benefit from greater uniformity in observations, which facilitates statistical analysis, and longer experimental sessions. We chose a lab-based experiment because our research questions focus on the *usability* of Code LLMs for beginning programmers and on their *processes*, rather than their educational outcomes. Specifically, the process of working with a Code LLM requires multiple, interdependent steps: (1) forming an intent, (2) crafting a prompt to describe the intent, (3) evaluating the quality of the LLM-generated code, (4) editing the prompt when the code is wrong, (5) editing the code manually, or (6) giving up and writing code manually (Figure 1). Our goal was to isolate processes (2) and (4).

Our study limits user interactions in order to isolate prompt writing and editing strategies. One key feature of our paradigm is that we automatically test the generated code. In most observational studies, programmers determine on their own whether the generated code is correct. This is itself an interesting process. However, studying this aspect of Code LLM interaction comes at the cost of studying prompt editing: if a programmer mistakenly accepts incorrect code, they will move on to the next task without editing. Prather et al. [78] report that many of their participants mistakenly accepted incorrect code. Beginning students are particularly likely to err in this way: they may struggle to understand generated code, and their lack of confidence in their own abilities may make them trust the automated system over their own judgment (an example of *automation bias* [18, 30, 32, 91]).

Finally, a key contribution of our work is its scale: we study 120 participants across 3 institutions and 48 programming tasks, while previous studies have had fewer participants and problems. We recruit participants from three U.S. institutions: an R1 university (Northeastern University), a small liberal arts college (Oberlin College), and a women's college (Wellesley College). This selection increases the likelihood that our findings will generalize across institutions. Our scale allows us to explore how diverse factors, such as prior non-curricular programming experience, first-generation status, and mathematics coursework, affect participant success. These kind of statistical analyses require large sample sizes and work best with even observations of participants and problems, which are challenging to obtain in course settings.

3.2 How to Describe Problems to Students: Input/Output Examples vs. Written Descriptions

A key design decision for studies of Code LLM interactions is how to present the task. In classroom environments, students are usually given instructions for what to program via written descriptions. This makes sense, given that the student's goal is to write code. However, natural language presentation poses critical issues for our key research questions. In our study, the goal is to write natural language descriptions of problems, *not* to write code. A core goal is to understand how students approach the natural-language-to-code task. If the task is presented in natural language, students may simply reuse this text rather than putting the task into their own words; our results would no longer measure beginning programmer

success, but instead expert description success. Prior work shows that this is a serious concern: in Kazemitabaar et al. [48]'s study of K-12 students, up to 49% of submissions for challenging problem categories were copied from the expert-written task description.

Even if participants do not directly copy a description, its wording could influence how participants describe the task. One challenge for beginning programmers is recalling and applying technical vocabulary; presenting them with a natural language description of the task might remind them of terminology that they would not have recalled on their own. This would endanger our goal of assessing beginning programmers' abilities to prompt code generation models, since in many natural settings, they would not have an expert description to rely on.

We therefore rely on a popular alternative for describing program behavior: input/output examples (Figure 3). Students also could reference the function name and parameter names. Our participants had taken CS1 classes where natural language descriptions are frequently accompanied by input/output examples (see Appendix A.1.2), making this a familiar way of communicating program behavior. Several CS1 courses, and some of the assignments used in our CS1 courses, go beyond this and require students to construct their own examples or even practice test-driven development [21, 27]. However, our study does not require students to write their own tests.

Avoiding natural language presentation is critical in order to study how beginning programmers describe problems in their own words. However, it comes with two risks. First, the input/output paradigm may increase task difficulty, since participants must identify the key pattern on their own. Although understanding natural language descriptions of coding tasks is not always easy for beginning programmers, it is likely easier than our input/output paradigm. Second, input/output examples run the risk of underspecification [35, 88] – there may be more than one program that performs the correct input-output mapping. To determine that the provided tests adequately described the problem, we confirmed that our provided test sets had 100% code coverage for a correct solution and performed mutation testing [46]. We also calculated participants' success using only the provided test cases: if the generated code passed the provided tests, it was deemed correct, ensuring that the problem presentation aligned directly with the feedback to the user.

We feel that these potential issues pose less of a risk to our key research questions than the copy/paste or word bias risks posed by a natural language presentation. Other researchers have also used an input/output presentation paradigm in studying beginner interactions with Code LLMs [19].

3.3 Problem Selection: Previously Seen Tasks vs. New Tasks

The natural language-to-code task requires participants to describe specific programming problems. Previous work exhibits varied approaches to problem selection, from a single challenging problem in Prather et al. [78] to three simple problems in Denny et al. [19] to a set of 45 problems in 5 categories in Kazemitabaar et al. [48].

Our main goal was to select problems at an appropriate level for students who had completed only CS1. Since our research questions focus on student prompting processes, not learning outcomes, we chose problems at a similar level to what participants might be able to code independently. Asking students to solve *new* or more complex problem types increases the likelihood that the Code LLM will generate unfamiliar or difficult to understand code, making the prompt editing process more difficult. We therefore adapted Python problems specifically from CS1 course materials at each institution. We made small changes to facilitate input/output testing or adjust problem difficulty. Appendix A.1 contains two examples of how source problems were adapted.

We selected 48 problems balanced across eight conceptual categories from CS1 (Figure 2), similar to Kazemitabaar et al. [48], but with more categories and problems. Each individual problem was assigned to 20 students; we balanced the experimental lists to control for ordering effects, so that each participant solved one problem in each category, and the average difficulty of each problem list was roughly the same. To facilitate difficulty and category coverage, previous CS1 instructors were asked to provide additional problems as needed. Problems such as `exp` (Figure 3), for instance, require students to only recognize that numbers in a list are being squared. Other problems ask students to remember complex data structures (e.g. lists, dictionaries), but not the specific Python syntax for them. We further discuss student understanding of the problems in Section 7.2 and Appendix B.2.

In order to study interactions between Code LLMs and students, it is important to select problems that cannot be trivially solved by a Code LLM without any natural language description. Very common functions (for instance, `shorten_url`) can be solved from a function signature alone, regardless of the accompanying description. To validate our problems, we first checked that the model could not solve problems from their function/parameter names alone and, if they could, edited the names accordingly. We also solved each problem using the Code LLM to ensure that a working natural language description existed. Finally, to address the nondeterminism of Code LLMs, we ran each validation check multiple times to obtain a stable estimate of these results (§5.2).

4 STUDY LOGISTICS

The previous section (§3) described our multi-institutional experimental design. In this section, we discuss the logistics of participant recruitment and executing the study.

4.1 Charlie Interface

We built a web application for the experiment called *Charlie the Coding Cow* or *Charlie*. Charlie presents one problem per page, displaying the function signature and several input/output examples (Figure 3a). Participants write natural language descriptions in a text box. When they submit a description, the Charlie server prompts Codex with the function signature and their description formatted as a docstring (Figure 4). After Codex responds, Charlie shows students the Codex-generated code and displays whether it works on the given input/output examples (Figure 3b).

Charlie does not permit participants to edit the generated code, since we are focused on natural-language-to-code interactions. If the code fails, they can retry the problem or move to the next problem. For retry attempts, we pre-fill the text box with their last prompt to make editing easier. Finally, after every final attempt

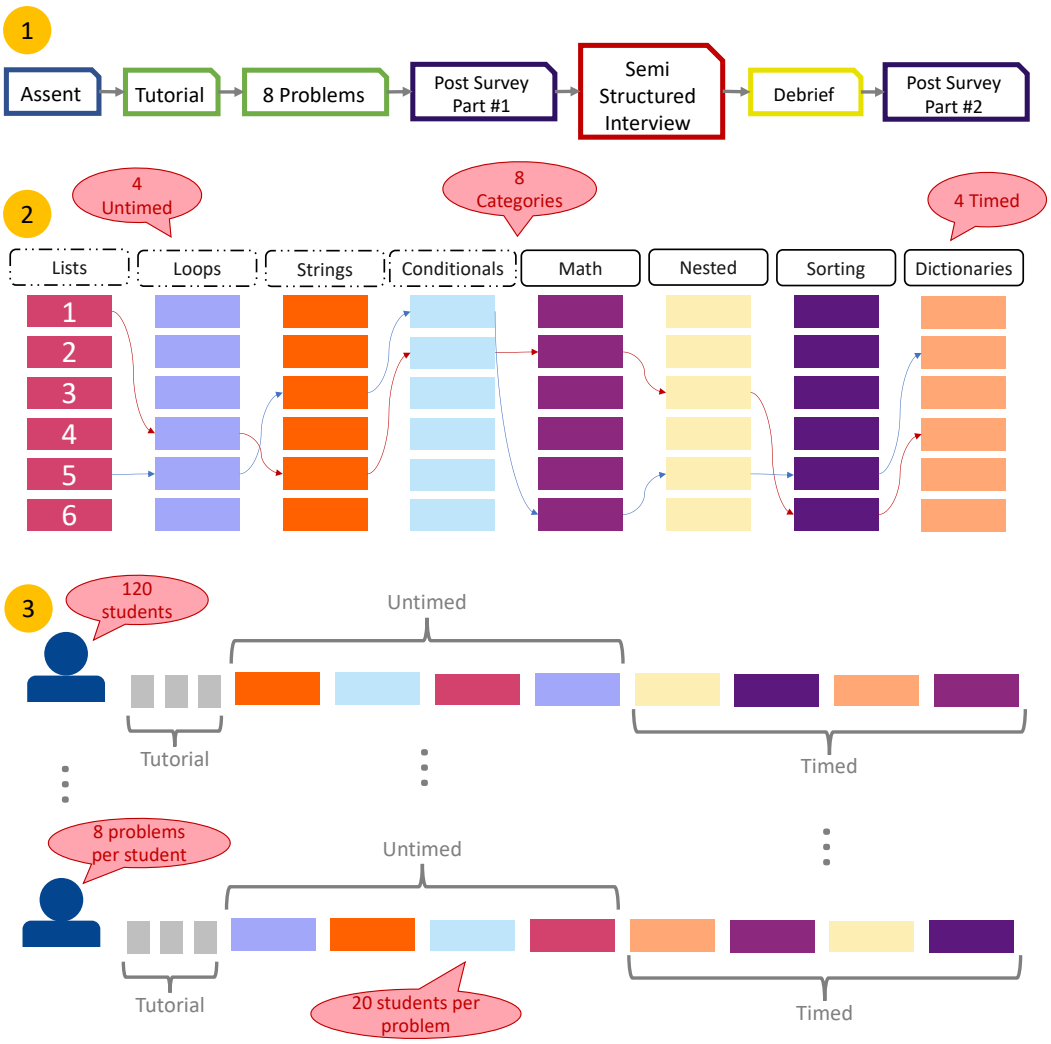


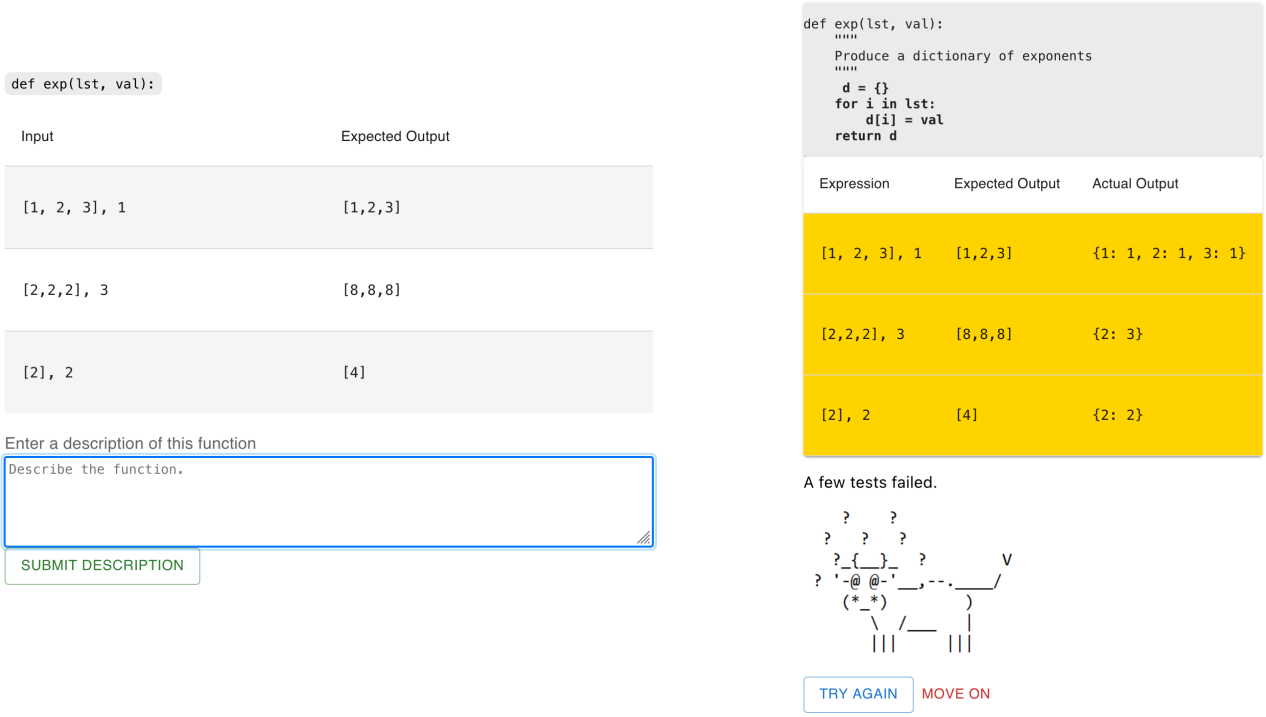
Figure 2: Study overview. (1) describes the overall student trajectory through the study. We split the post survey into two sections, divided by the semi-structured interview, to delay collecting demographic information to prevent self-bias. (2) outlines the 8 problem categories (4 timed versus 4 untimed) and the 6 problems per category. Students took individual trajectories through one problem in each category, as shown by the thin arrows. (3) showcases an example trajectory for students through the problems. Students spent, on average, 42.6 minutes (SD=10.6) completing the study, with an average of 26.6 minutes (SD=9.1) on the untimed section and 15.9 minutes (SD=3.3) on the timed section.

at a problem, Charlie presents two forced-choice questions with thumbs-up / thumbs-down answers: *Did Charlie generate correct code?* and *Would you have written this code yourself?*. We included these questions to gather information about student perceptions of code style, since the model may produce working code, but in a style that is unfamiliar to students.

Each student worked with Codex to solve 3 tutorial problems and 8 main problems. We used the Charlie character to provide distance from any AI system that students might already know. This suggested a representation that was not human and not robotic. Charlie also provides visual feedback: Charlie animates a "thinking" position while Codex generates a completion and appears in different forms when the code does or does not pass all tests. We made this design choice to mitigate frustration with waiting for the model to generate code, a source of annoyance in prior studies of Code LLM interactions [69].

4.2 Model Choice

When we began piloting in November 2022, the most capable Code LLM was the largest Codex model from OpenAI, `code-davinci-002`. Although `code-davinci-002` was first released in 2021, on established Python programming benchmarks, it remains as good as `gpt-3.5-turbo`, which is the model presently used for GitHub Copilot's inline completions [104], the free version of ChatGPT, and



(a) An example task posed to a participant. The interface displays the function name and several input/output examples. Participants write and submit a description in the text box. During our study, 85% of students who attempted this problem wrote a successful description after a single CS1 course.

(b) We run expert tests automatically and highlight ones that fail. Students are then able to either edit their description by pressing "Try Again" or move on to another problem.

Figure 3: The *Charlie the Coding Cow* interface.

several other commercial products. Specifically, `gpt-3.5-turbo` and `code-davinci-002` score 48% and 46% respectively on the HumanEval Python programming benchmark [11, 74], the most commonly used Python benchmark for Code LLMs. Since we started our study, several other LLMs have also appeared, including non-proprietary LLMs that are better for reproducibility (§9.5). The best open models perform comparably to `code-davinci-002`; for instance, CodeLlama (34B) achieves 48% on HumanEval [85]. This suggests that the model that we use is as capable at code completion as newer models used in practice.

There are larger models that are more capable, such as GPT-4, which achieves a HumanEval score of 67% [74]. However, GPT-4 is significantly slower and higher latency than the alternatives, and low latency is essential for LLM code completion to be acceptable to users [69]; if participants have to wait more than a few seconds for the generated code, their frustration might lead them to move on rather than re-attempting the problem.

For consistency, we used the same Codex model throughout the study (`code-davinci-002`). It is important to note that Code LLMs perform best when their output is sampled; consequently, the model may produce different programs for the same prompt. We generated output using best practices for hyperparameter and sampler settings [13].

4.3 Participants

We recruited 40 participants from each institution (n = 120). Eligible participants were at least 18 years old, had taken CS1 at their institution between Fall 2021 and Spring 2023, and had not completed any subsequent CS courses. We recruited participants from March to July 2023 until reaching our sample size of 120. The pilot and main study received IRB approval.

Care for Participants. Our study design sought to balance obtaining accurate data with addressing potential discomforts and power dynamics. Potential discomforts for participants included frustration regarding their inability to complete a task, which could reinforce negative perceptions of self or CS. In the tutorial, we emphasized that our goal was *not* to evaluate their programming skills, but the collaboration with Charlie. Students were allowed to move on from a problem at any time, resulting in a variable number of attempts per problem.

We took several steps to address potential power dynamics between students and their professors. Recruitment was done through an interest form distributed by other faculty or staff. Scheduling was performed by a researcher at another institution. Finally, research sessions were never run by a professor at the same institution as the participant.

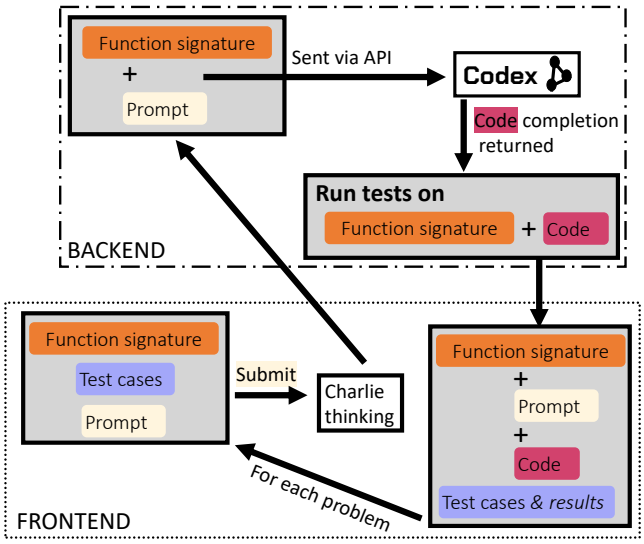


Figure 4: An overview of the experimental platform. For each problem, the frontend provides the participant with the signature and tests and asks them to write a description (prompt). This is then relayed to the backend, where the signature and prompt are sent to Codex via the API. The code completion from Codex is then run on our pre-defined tests. Finally, the results of running the tests and the code completion are presented to the participant in the frontend interface.

4.4 Study Execution

The study was conducted over Zoom with audio and video recording. Participants signed informed consent material ahead of the experiment and assented at its start. They were compensated with a $50 gift card for the estimated 75-minute study.

Main Task. Figure 2 (1) outlines the full study design. Students completed 3 tutorial problems to get familiar with the interface and see some possible Codex responses. We supplied participants with a working prompt for the first tutorial problem, then gave them a difficult problem so they could see a failure, and a final easy problem to solve independently.

The main experiment consisted of 8 problems in two blocks, the first untimed, the second timed. In the second block, students were limited to 5 minutes per problem. We included both timed and untimed blocks in order to balance the need to bound study duration with the desire to observe complete prompt editing cycles.

Participants were randomly assigned experimental lists, balanced by difficulty, using a modified Latin Square design. Four authors independently assessed the difficulty of writing prompts for each problem; we averaged these scores and developed six roughly equal lists (Figure 2).

Post-task Interview and Survey. After the main study, students completed a two-part survey, a semi-structured interview, and an optional debriefing session (Figure 2 (1)). The semi-structured interview was interleaved between two survey blocks to mitigate question ordering and priming biases.

The first part of the survey was designed to study student perceptions of Charlie and of AI more broadly. We adapted validated scales from previous work to understand student perceptions of the usability, trustworthiness, and friendliness of Charlie [7, 20, 51, 99] and the mental workload of the task [36].[2] We were also interested in whether students' ability to come up with effective prompting strategies might correlate with fixed versus growth mindsets about computing; we drew on Gorson and O'Rourke [33] to measure this.

The semi-structured interview asked 8 questions covering student editing processes, what they found hard or easy, how they envisioned their interactions with Charlie, and how they imagined Charlie worked. The specific questions were directly inspired by our overarching research questions. Researchers followed a standing script to ask each question - there are a total of 5 missing question responses across the possible 960 interview datapoints, likely due to researcher error or time considerations. In the optional debriefing, we explained the experiment and how Code LLMs work.

The second part of the survey focused on participants' backgrounds and demographics. These were the last questions of the study to mitigate possible stereotype threat [72]. For questions related to identity (e.g., gender, race, spoken language at home), we followed best practices and solicited responses via open text boxes [92]. We also asked questions about students' CS1 performance, experience with programming outside of CS1, high school & educational background, math background, major, and class year.

Pilot Study. In late 2022, we ran a pilot study with 19 participants to assess the study design and usability of the interface. Pilot participants were recruited from the same three institutions as in our main study, but were students who had taken more than one CS course. This small pilot allowed us to make sure the web platform was working correctly, identify any problems with specific tasks, refine our time estimates, and assess the quality of the automatic transcriptions of the interview recordings produced by otter.ai.[3] During the pilot, we identified one problem with ambiguous test cases, which we changed before the main study. Pilot participants solved an average of 5.5 out of 8 problems (an Eventual Success Rate of 68.8% using the metric described in §5.2).

Because the average pilot participant took 53 minutes, we increased the time estimate and compensation from $30 for 60 minutes to $50 for 75 minutes for the main study. We also added a hidden time limit to the first block of questions in case participants spent more than 50 minutes on this portion of the study; this issue never arose in the main study.

5 ANALYSIS

This section presents the analysis framework for §6, §7, and §8. We take a mixed-methods approach to this work.

[2] In some cases, we removed questions that were not relevant to our study to keep the survey length manageable for participants. Details available via our Supplemental Materials at https://doi.org/10.17605/OSF.IO/V2C4T.

[3] https://web.archive.org/web/20231205001012/https://otter.ai/

5.1 Evaluation Plan

Qualitative analysis. We collected three types of data which lend themselves to qualitative analysis: (1) information about student experience and demographics, (2) free-response questions about future use of Charlie, and (3) semi-structured interview responses. We employed both inductive and deductive open coding towards consensus. Our aim was to identify common themes present in this specific dataset, rather than to develop a theory. Two researchers with previous qualitative experience conducted the analysis; Section A.2 contains details of the coding methodology. We present selected quotes from the surveys and interviews throughout. Quotations have been lightly edited from the automatically generated transcripts. This includes addressing grammar/punctuation, removing speech errors or filler words, and avoiding the disclosure of any identifiable information. Each participant's quote is accompanied by a pseudonym assigned to them during data collection.

Statistical analysis. We perform statistical testing with a significance level of α=0.05 in order to determine whether observed differences in response measures are statistically reliable. For comparisons between two groups, we use Student's t-test. For comparisons between multiple groups, we perform ANOVAs; in cases where there is no natural reference group, we use Tukey HSD tests to explore pairwise differences. We report Pearson's r for correlations between continuous variables and Kendall's τ for correlations between continuous and ordinal variables. Where we are interested in multiple potentially interacting variables, we fit linear mixed-effects models with maximal random effects for participants and problems using the lme4 package in R [8].

5.2 Measures of Success

There are several ways to measure success when evaluating the natural-language-to-code task. The *success rate* is the fraction of all attempts on which the model generates a working program. Therefore, a participant who takes several attempts to solve a problem will have a lower success rate than another who succeeds in one try. We might also ask whether a participant is ever able to solve a problem; we refer to this as the *eventual success rate*. This metric considers only the participant's final attempt at each assigned problem. The eventual success rate metric is likely specific to this paper, as closely related work [19, 48, 78] studies different notions of success or does not permit controlled, repeated interactions.

Although success rates measure the correctness of the code that students saw during the experiment, LLM generation is non-deterministic.[4] Therefore, studying success rates can be misleading: a participant may have just been lucky with a bad prompt or unlucky with a good prompt. For this reason, we also employ an alternative metric called *pass@1*, which accounts for non-deterministic generation [13]. Since the debut of Codex, pass@1 has become the standard metric used to evaluate LLMs on the natural-language-to-code task, including GPT-4 [74], Code Llama [85], and other models [29, 59, 73].

Given a natural language prompt, *pass@1* [13] is an estimate of the probability that the LLM will generate working code in one attempt. In the LLM development literature, the accepted best practice for computing pass@1 is to query the LLM 200 times for the same prompt and test every generated program [13, 29, 85, 102]. Sampling 200 generations for all 2,000+ prompts generated as part of this study would be very expensive with the Codex API. Instead, we use a recently released open Code LLM called StarCoder [59] that is nearly as capable as the Codex model on Python benchmarks. Pass@1 with StarCoder will be slightly lower than Codex success rates because of model differences. However, pass@1 is a more stable measure of whether a prompt will succeed than success rate. We use pass@1 for the bulk of our analyses.

[4]Greedy generation is significantly worse for coding tasks than non-deterministic generation [13].

5.3 Positionality

All authors were affiliated with the institutions from which participants were recruited (Oberlin, Wellesley, or Northeastern) at the time of the study; we range from undergraduate students to tenured faculty. We developed the problem lists, problem difficulty ratings, and other elements of the study design within a shared educational context. The last three authors are course instructors for CS1. As described in §4.3, significant care was taken to address power dynamics between participants and researchers. Some authors also contribute to the development and evaluation of open-source Code LLMs. Overall, the potential incentives for the research team are complex, as we approach this work as both educators and researchers. We aspire to a neutral perspective on Code LLMs, while attempting to center the student experience.

This research studies students at three selective higher education institutions in the United States. Therefore, while we are able to generalize beyond a single CS curriculum, the educational context is specific: our findings may not generalize to other settings (e.g., community colleges, K-12 education) or cultural contexts.

6 RQ1: DO STUDENTS SUCCEED AT PROMPTING CODE LLMS WITH NATURAL LANGUAGE?

In this section, we present how well students do on our Code LLM prompting task and address RQ1: do students succeed at prompting Code LLMs with natural language? We explore differences between students that are linked to their ability to successfully describe problems to Code LLMs.

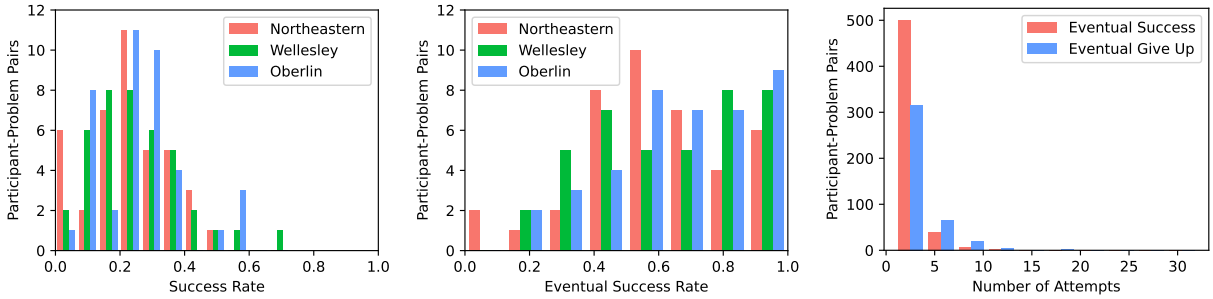
6.1 Basic Findings

Figure 5 presents the distribution of participants' success rates and eventual success rates. The average participant solved 4.7 out of 8 assigned problems. The mean eventual success rate (57%) is not high, and the mean success rate (24%) is even lower, since it decreases with every failed attempt. We find no significant institutional difference for either measure of success.

Participants often submitted a large number of failing attempts (Figure 5d): 153 problems (aggregated across participants) required three or more attempts. In fact, one participant succeeded at a problem only after 32 attempts; another gave up after 26 attempts. These results suggest that low success rates are not due to a lack of participant effort. Participants struggled to write natural language prompts for the LLM, and often achieved success only after many

Institution	Mean pass@1	Success Rate	Eventual Success Rate
Oberlin	0.23	26%	61%
Wellesley	0.23	25%	57%
Northeastern	0.20	23%	54%
Overall	0.22	24%	57%

(a) Mean values of different measures of success.



(b) Success rates. **(c) Eventual success rates.** **(d) Attempts.**

Figure 5: Basic measures of student success at the natural-language-to-code task. *Success rate* is the fraction of all attempts by a participant that succeed. *Eventual success rate* is the fraction of last attempts at a problem by a participant that succeed. Pass@1 resamples the LLM several times to estimate the probability of success. We present these measures by institution. Figure 5a presents the means. Figure 5b and Figure 5c show the distribution of (eventual) success rates. Eventual success rates are higher than success rates, which is to be expected: Figure 5d shows that many students make several attempts at a problem before an eventual success or give up.

Self-Reported Background	N	Mean pass@1
International	92	0.23
Domestic	27	0.22
First-generation college student	23	0.17
Not first-generation	96	0.23
Attended private high school	38	0.22
Attended public high school	76	0.22
Raised Monolingual in English	49	0.22
Raised Monolingual Not in English	27	0.20
Raised Multilingual Including English	41	0.24
Raised Multilingual Not in English	2	0.22

Table 1: Self-reported high school, language, and family background.

repeated attempts. The challenging nature of this task is supported by comments from the students themselves (§7.1).

6.2 Do Participants Find the Task Challenging?

In the post-survey, participants completed four items of the NASA TLX [36]. Overall, students found the task mentally demanding (Table 2). The questions about mental demand (Q1), time pressure (Q3), and their own performance (Q4) correlate inversely with success rate. Students whose success rates were lower generally rated the task as more demanding (Kendall's τ=-0.16; p=0.02); were less likely to say they were successful (Kendall's τ=-0.4; p<0.0001); and reported higher levels of stress and insecurity (Kendall's τ=-0.27; p<0.0001).

6.3 Who Succeeds at the Task?

Using data from the post-survey, we analyze the relationship between pass@1 rates and previous knowledge, prior programming experience, and demographics (see Table 1 for a summary of demographics). We find only two statistically reliable differences (see Appendix, Table 11 for the full statistical analyses):

- **Prior programming experience:** About 1/3 of participants had no programming experience outside of CS1. The remaining participants had taken pre-college programming courses (24%), were in the next CS course (21%), or had coding experience outside of classes (29%). There is a statistically reliable difference (t-test; p = 0.02) in pass@1 for students who have only coded in CS1 (0.17) versus those with additional experience (0.24).
- **First-generation college students:** 19.1% of participants identified as first-generation college students. We observe a statistically reliable difference in pass@1 for first-generation participants, who struggle more with the task than others (t-test; p=0.04).

We examined other factors, but found no significant difference in pass@1 rates:

- **Math courses:** All but one participant had taken at least one college math course and half had taken 2+ courses. Single variable calculus was the most common math course. There is no statistically reliable difference between participants who had or had not taken 2+ math courses (t-test, p=0.42).

Abbreviated Question	Scale (1 to 7)	Mean
How mentally demanding was the task?	Very low->Very high	4
How hurried or rushed was the pace of the task?	Very low->Very high	3.3
How successful were you?	Perfect->Failure	3.6
How insecure, stressed, or discouraged were you?	Very low->Very high	3.1

Table 2: Mean NASA-TLX ratings [36].

Thematic Codes	N
Charlie Doesn't Understand Me	91
Issues With Generated Code	59
Student Struggles	41
No Problems Mentioned	10
Issues with Study Platform	10
Issues With Experimental Design	7
Easier To Write Code Myself	7

Table 3: Thematic codes emerging from responses to *What kinds of problems or issues did you run into working with Charlie?*

- **Computing intensive majors:** 42% of participants were pursuing computationally intensive majors. We observe identical pass rates for both computing and non-computing majors.
- **International students:** International and U.S. domestic students had similar pass@1 rates.
- **Household language:** Our participants reported growing up in households where a diverse set of languages were spoken: only English (40.8%), English and other languages (34.2%), and without English (24.2%). We were surprised to find that pass@1 did not reliably vary by childhood language. However, all participants were from selective U.S. institutions that require fluency in English, regardless of childhood language exposure.
- **Public vs private high schools:** 1/3 of participants attended private schools; this had no impact on pass rates.

7 RQ2: WHERE DO STUDENT DIFFICULTIES COME FROM?

Having shown that students find it hard to prompt a Code LLM in natural language (§6), we explore why. In this section, we present quantitative and qualitative results that address RQ2: when students struggle with the task, where do the struggles come from? What are the most challenging aspects of the natural-language to-code task?

7.1 What aspects of the task do students say are hard?

In the semi-structured interview, we asked participants to reflect on challenges and issues they encountered. Three common themes emerged: difficulties in getting Charlie to understand them; issues with the generated code; and issues stemming from students' self-reported lack of knowledge or skill (Table 3).

Charlie Doesn't Understand Me. The most commonly raised issues related to Charlie's understanding of prompts (n=91); we divided these into subcodes. One of the most common of these was the sentiment that Charlie failed to understand good descriptions (n=23). For instance, redCoyote commented, *"It was definitely difficult to have a concept of what you wanted written in your head, and then feel like you're articulating it well, but having it not work properly."* Similarly, aquaLadybug reports feeling helpless when a good prompt didn't succeed: *"if I was saying it [...] how I thought [...] is the best way to say it, but it still wasn't working, I had no idea where to go from there."*

Issues with Generated Code. Another major theme was issues with the generated code. Many commments related to perceived bugs in the generated code or difficulty debugging (26%). Students also mentioned finding the model's randomness frustrating (8%). khakiBee was alarmed to find that resubmitting the same prompt could generate different programs, commenting *"You feel like you've made progress, and then because it did a different thing the next time, it's like, what do I change? I'm trying to change what I give to the cow. And then that should change what the cow is doing. But if I'm not changing anything, why is that changing?"* Some students also experienced the opposite issue: despite changing their descriptions, the model generated the same incorrect function repeatedly. purpleCarp commented, *"Sometimes I changed my [...] description and it just repeated the code the same. And it's just very frustrating"*. This highlights the difficulty of working with stochastic models: students expect the model output to be faithful to their descriptions.

Student Struggles. Participants also reported issues stemming from their own lack of knowledge. 10% of students reported difficulty understanding a problem, and 8% reported difficulty in understanding generated code. yellowChipmunk said, *"sometimes with the code, just given my knowledge, that's not necessarily the way I would go about coding the code. But I think to even understand it, I would have to know what the code is trying to do, which takes more time than me just trying to reword what I said"*. A handful (n=4) reported that forgetting terminology made it hard to write prompts.

7.2 Which Problems Do Students Say Are Hard?

Some categories of CS1 problems may be harder to solve with Code LLMs, either because the concepts are difficult or because they are difficult to describe. We examine pass@1 and eventual success rate by category as well as interview responses about which problems were challenging and easy.

We find that pass@1 and eventual success rates both vary by category (Table 4). We fit a binomial mixed-effects model to prompt success (1 if the prompt succeeded; 0 otherwise), with fixed effects of category, institution, and their interaction, and random effects

Category	Mean pass@1	Mean Eventual Success Rate	Student Difficulty Ranking
Sorting*	0.09	33%	1 (Hardest)
Dictionaries*	0.17	43%	2
Nested*	0.30	68%	6
Math*	0.16	54%	4
Loops	0.13	52%	3
Lists	0.18	61%	5
Conditionals	0.33	73%	7
Strings	0.26	74%	8 (Easiest)

Table 4: Pass@1 and success rates by problem category. Each category has six problems, and an equal number of students attempted each problem. The starred (*) problems were timed. Student Difficulty Ranking is done by ordering mean Eventual Success Rate from least to greatest, as that provides as measure of what percentage of students successfully solved a given task.

of problem and participant (see Appendix, Table 12). A statistically reliable difference in success was observed only for Sorting problems, which were the most challenging (p=0.045). Participants from Oberlin struggled more in the Nested category compared to other students, but the effect is not statistically reliable (p=0.063).

Interviews provide insight into their post-task perspectives. The most commonly mentioned easiest category was Math (n=21), whereas the most common for hardest was Nested (n=19), followed by Dictionaries (n=14). These do not match the ranking in Table 4, suggesting a disconnect between student performance and perceptions of difficulty.

A common theme that emerged related to the challenge of putting understanding of the problem into English (n=44). CRIMSONVOLE said, *"the ones that had huge lists of like, strings, and integers, were really hard to solve, because they were really hard to describe for me."* We differentiated this code both from students' ability to identify patterns (n=35) and their ability to write the code without Charlie (n=8). The opposite code, Easy to Describe, applied to 36 responses from the easiest question: *"I felt like time ones because they're pretty straightforward. They're like [...] exercises that we do in my Intro CS class. And so I guess it will be easier for me to word, the description or my thinking process, like I guess that might be easier."* (YELLLOWWEASEL).

Three codes that related to student's lack of knowledge emerged, with 27 responses (see §7.4 for more perspectives).

7.3 What Role Does the Model Play?

LLMs can fail in surprising ways. We now explore the kinds of model failures that participants encountered.

7.3.1 Syntax errors. Contemporary Code LLMs generally produce syntactically well-formed programs. However, 5.5% of student prompts led to Python syntax errors. We manually examined and categorized them:

- 27 generations: Codex produces degenerate, repetitive text [43] or Python 2 print statements. These are model failures.
- 81 generations: Codex could not generate a complete function within the 256 token limit (≈800 characters). Our problems are simple enough to be solvable in far fewer tokens, so increasing the token limit is unlikely to help.
- 88 generations: Codex generates incomplete code after a complete function, even with standard stop tokens.

The latter two categories arise from a trade-off in system design: the first when the interface does not request enough tokens from the Code LLM; the second when it requests so many that the model generates extraneous additional code. Although these errors are infrequent, they are hard for students to deal with. In 22.4% of these cases (n=44), students gave up after seeing the syntax error.

7.3.2 When the Model Produces Different Programs From the Same Prompt. Codex is best at coding when its output is sampled (§4.2), but this stochasticity can frustrate students trying to modify prompts. In 107 cases (4.2%), a student submitted a prompt several times, and in most of these cases, Codex generates a new completion. A few of these are trivially different (e.g., different variable names), but most (n=86) are different functions. Some students pointed this out in the interview – BEIGEHALIBUT noted that they *"usually would run a couple times, because Charlie is not very consistent with the answers. And sometimes it works. Sometimes it wouldn't work."*

7.3.3 When the Model Produces the Same Program Despite Changes to the Prompt. When the Code LLM produces an incorrect function, and a user edits their prompt, their intent is to have the LLM produce a different—hopefully correct—function. Frustratingly, this does not necessarily happen: sometimes the model repeatedly generates the same code despite edits to the prompt. We observe many instances where this happens (104 submissions, 11% of total): it occurs in most problems (36 of 48 problems) and is encountered by a majority of students (72 of 120 students). This often leads students to give up. In fact, out of the 340 problems where students gave up, 70 were cases where the participant edited the prompt and the LLM repeatedly generated the same code.

7.4 What Do Students Do When They Encounter Unfamiliar Python?

Code LLMs are trained on online repositories of code and may generate code using language features that students have not seen before.

New Python Constructs. In their interviews, some students (n=5) report issues understanding code due to unfamiliar language features. OLIVEBEAR comments about the lambda construct for anonymous functions: *"I've only ever seen [it] in passing. And so if that hadn't worked, I wouldn't have known what the problem was because I myself don't know how to use that operator."* Others mentioned `map`,

Completion	`def exp(lst, val):` `  return [i ** val for i in lst]`
Question	Is this code you would write yourself?
Student Responses	Wellesley: Yes, Oberlin: No

Figure 6: An example code completion for the problem exp – this was generated by multiple different prompts. The completion was rated differently by Oberlin and Wellesley students, likely due to the list comprehension.

`replace`, and `try`/`except`. List comprehensions are an interesting case because Wellesley teaches them, but Oberlin does not. When asked about generated code with list comprehensions, 9/24 (37.5%) Oberlin students indicated that it is similar to code they would write themselves, compared to 20/33 (60.6%) Wellesley students. Some students responded differently to the same completion (Figure 6).

Ratings of Final Completions. Students evaluated the correctness and naturalness of the final completion for each problem, producing 960 responses. For correctness, 61.8% of the time students indicated that Charlie's code was correct; the majority (543; 91%) are cases where all tests passed. However, naturalness responses were more mixed. Students indicated that Charlie's code was like code they would write themselves only 58.3% of the time. 78.6% of such responses were made when the code passed all tests. Responses to these questions might diverge when the model generates correct code that is unfamiliar or approaches a problem differently, as well as in cases where the model's code is incorrect, but looks familiar to students.

8 RQ3: STUDENTS' MENTAL MODELS AND PROCESSES

This section addresses RQ3, presenting results related to participants' perceptions of the task, their mental models of Charlie, and their strategies for writing prompts.

8.1 How does Charlie work, according to students?

In interviews, students were asked how they thought Charlie worked (Table 5). Comments fell into two broad themes: descriptions of Charlie's knowledge, and descriptions of Charlie's processes.

Processes. Comments in the Translation theme (n=13) described Charlie in terms of a machine translation process (FUCHSIABEAVER: *"I thought of him as like a translator, like between English and code*"). Comments in the Sequential theme (n=13) described Charlie as working line-by-line through their prompt. This is plausible but incorrect: Code LLMs condition on the entire prompt at once. This mental model might lead students to focus on individual sentences, rather than how their prompt works as a holistic description. One student actually changed their mental model while answering: *"it looks like he went line by line. Wrote some code for each line that makes sense to him [...] Actually, no, I think he takes in the whole prompt and [...] figures out what to do with the prompt. Because I do remember [...] there were a couple where I give a paragraph and then he returned a line of code, which makes me think that he wasn't going line by line."* (KHAKICLAM).

Thematic Codes	N
Knowledge: Keywords - General	30
Knowledge: Keywords - Database/Dictionary	16
Knowledge: ChatGPT	17
Knowledge: Internet Data	12
Knowledge: Intermediate Representation	4
Knowledge: Copilot/Codex	2
Process: Sequential	13
Process: Translation	13
No Guess	13
N/A	24

Table 5: Thematic codes emerging from responses to *How did you imagine that Charlie was working?*

Charlie's Knowledge. Most students hypothesized that Charlie relies on keywords (n=46). A large group of students (n=30) had a vague keyword mental model. For instance, *"I guess he probably looks for keywords, "if" and "else" and key coding words, Python words, and he probably has a knowledge of English"* (WHEATOTTER). Another group (n=16) outline a more specific keyword lookup model, where Charlie uses keywords to retrieve relevant code from a dictionary or database. For instance, LINENBOBCAT described Charlie as *"using the code words, and doing it sort of line by line and trying to work from what was given and writing those words with what, like in a directory or some sort of data file, understanding which ones matched up to which functions and which commands."*

Students with this mental model emphasize the importance of using programming terminology, since they think Charlie may not be able to retrieve code without the right keywords. Some students develop this mental model after observing that their prompts succeed when they use coding words: *"I noticed that if I put in more like, computerized words, I almost had a bit more control. At one point, I forgot to mention that the function returns something. So then when I mentioned that it returned something he put in a return statement. So that felt like very, like logical to me. [...] Charlie's looking for words that kind of line up with different functions, built in functions, and using those."* (TANMINNOW). These students correctly observe that sounding like a programmer is important, but explain this with an incorrect mental model.

Some students did correctly identify Charlie as similar to an LLM such as ChatGPT (n=17) or Copilot/Codex (n=2). Success rates for this group were slightly higher (0.27 versus 0.22; p=0.03).

8.2 What strategies do students develop?

The first two semi-structured interview questions asked students about their strategies for writing and editing prompts. We find that students do not have a clear understanding of how models work and that their incorrect mental models appear to affect the strategies they develop for prompting in ways that might be unproductive.

8.2.1 Editing processes. Over a third of students (n=48) mentioned adding detail to their descriptions when they did not succeed (Table 6). Some students mentioned clarity as a goal in adding detail,

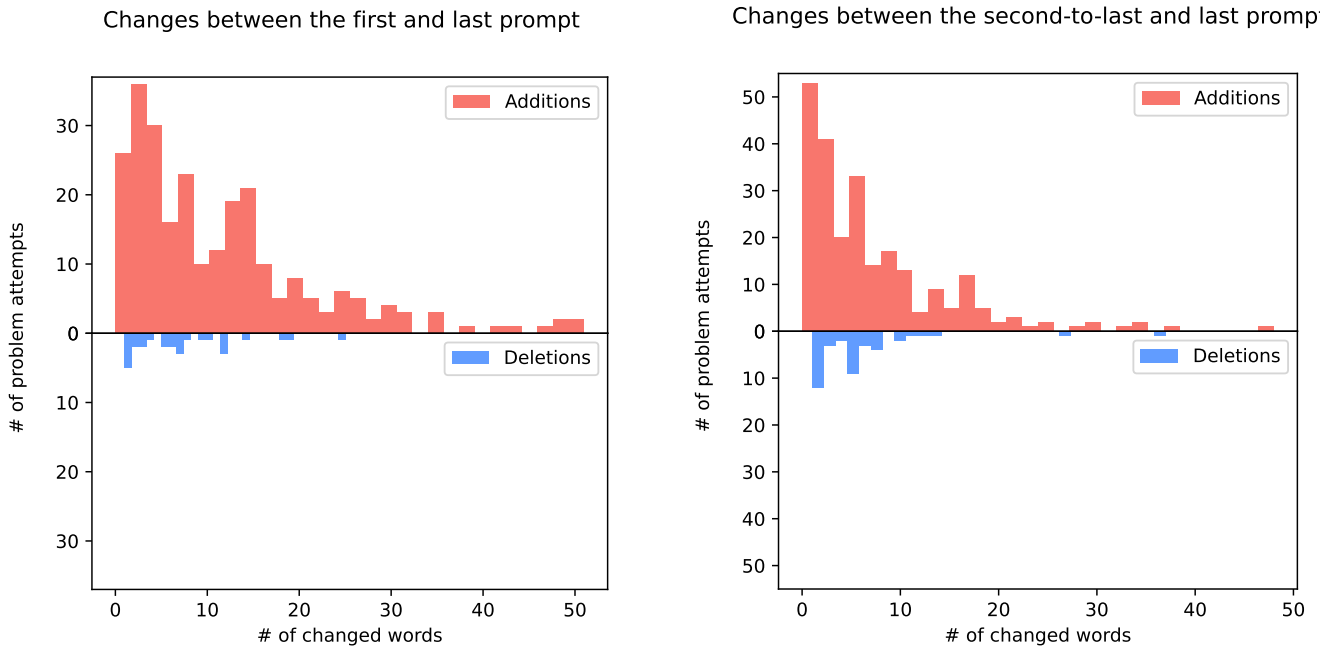


Figure 7: Histograms of the 282 prompts which lead to successes after 2 or more attempts. These represent trends in how students edit prompts. The figure on the (left) shows the number of words changed between a first prompt and last prompt. The figure on the (right) shows the final change that produces a successful final prompt.

Thematic Codes	N
Added Detail	48
Looked at Tests First	30
Looked at Code First	29
Added Coding Language	21
Comment Not Relevant	16
Looked at Code and Tests Together	8
Reread the Problem	7
Reordered Prompt	5
Removed Detail	4
Ran Prompt Again	3
Fixed Grammar	2

Table 6: Thematic codes emerging from responses to *What did you do when you wrote a description, pressed Submit, and it did not work? Describe the steps you took to edit your description.*

like FUSCHIABAT: *"I will go back and try to change the wording to make it more clear, and then try it again. And see if that changes anything. And then just try to repeat that process until it works."* Others noted that their descriptions needed additional detail because they did not originally fully describe the problem, or as PLUMBEETLE puts it, *"I forgot to uppercase Aspen. And that was just my silly mistake. And I will just go back and edit or add changes that I want to add and wish it's gonna work the next time I guess."* Considering participants' edits quantitatively confirms the popularity of adding detail. When we consider pairs of prompts that ultimately succeed, we find that students, on average, add 9.44 words (SD = 11.34) between their first and last prompt, and 5.36 words (SD = 8.87) between their penultimate and last prompt (Figure 7).

While adding details was the most common approach, participants mentioned other strategies, such as reordering (n=5) or removing detail (n=4). There are also eight attempts where rerunning the same prompt resulted in a success; we discuss these cases in §7.3.

Students looked in different places for insight into how to edit their prompts. Some considered the generated code first (n=29), some the tests (n=30). Others considered both (n=8) or reread the problem (n=7).

8.2.2 Strategy changes over time. Participants had a range of responses about how their prompting processes changed over time. Some students indicated that they never really developed a process (n=13), while others (n=14) discussed actively testing and adapting to Charlie's capabilities: *"I first [...] was kind of seeing what vocabulary Charlie knew. Like if he knew computer science terms, or if I had to be less computer science-y"* (BEIGEBASS).

We present key trajectories in Figure 8. Overall, we observe a range of reported experiences. Some participants reported starting more human-like and ending more technical (Pythonic), while others said the opposite. For instance, TOMATOBEETLE reported, *"To begin with, I was using less technical terms and then using more computer science terms near the end. I was thinking that would make Charlie work better, but there wasn't really any evidence behind that"*,

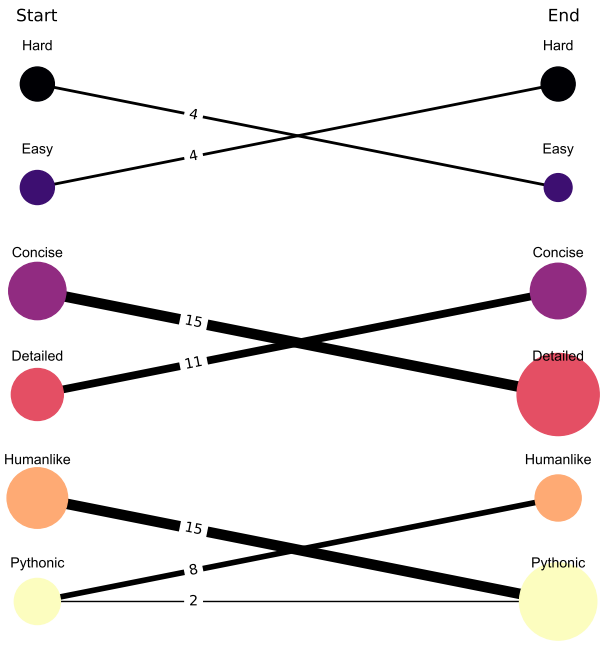


Figure 8: Visualization of how students describe their editing trajectories. The left nodes represent how students described how they began their process. The right nodes represent how students described how they edited prompts at the end of the study. The codes are presented in pairs - Hard versus Easy, Concise versus Detailed, Humanlike versus Pythonic. Only trajectories between pairs are visualized. The size of the nodes is proportional to the total number of students who described their Start or End within that code.

while GRAYRABBIT said, *"I kind of treated it like I was just coding but saying things I would like use kind of like if statements and integers and stuff. But towards the end, I tried to focus more on how I could say what was going on at a higher level, so using more plain language versus specific coding language."*

A large group reported that their prompts became more detailed (n=35) and/or more technical (n=31), mirroring the finding above that students typically add detail when editing. For instance, TANBAT reports, *"My initial process was just to figure out what the code is doing and then just write generic descriptions, like without any coding language inside of it. But then when I saw that Charlie kept having problems, I started to go to more coding language."* However, others took the opposite approach, and ended the study writing more human-like (n=11) or concise (n=16) descriptions.

8.3 Do Students Get Better at Prompting Over Time?

It is easy to argue that programming by prompting a Code LLM with prose is more natural than directly writing code and that Code LLM prompting is easy to learn. But how easy is easy? We investigate whether students improve at prompt writing over the course of the study. We explore this by comparing success rates for (1) students who attempted the problem first with (2) students who attempted the problem last. Our experiment design ensures that there are exactly 5 students who attempt each problem first and five more who attempt it last. We find no significant difference in success rates between the two groups, indicating that students do not observably improve at prompting within the 75 minute study.

8.4 What do students think about Charlie?

One of the most consistent findings in work on how experts use Code LLMs is that users enjoy using models [69, 105], even when no concrete productivity or correctness benefits are observed [97, 102]. However, near-novices exhibit different motivations and relationships to technology than expert programmers. This makes it important to investigate how non-experts feel about these systems.

8.4.1 Charlie's competence and reliability. The post-task survey asks participants several sets of questions related to their perceptions of Charlie. They completed 5 items from Bartneck et al. [7] adapted by Wang et al. [99] and Druga and Ko [20] for non-robotics use. Participants generally give Charlie middling ratings for knowledge and competence. Participants take more extreme positions on Charlie's persona, in opposite directions: they rate Charlie as both friendly and machinelike. Students who experience lower success rates find Charlie somewhat less competent, but do not seem to find Charlie less friendly (Table 7). Students also completed 5 items from Körber [51]'s trust of automation survey. Overall, students see Charlie as somewhat reliable and somewhat interpretable (Table 8). Students with higher success rates tended to rate Charlie as less error prone, easier to understand, and more reliable.

8.4.2 Would they use Charlie? The post-survey asked about students' attitudes toward hypothetically using Charlie in (a) the CS1 course they completed and (b) their own future programming practice. We used a thematic analysis approach to analyze this data, as with the interview data (see Appendix A.2 for more details).

Overall, two-thirds (n=83) stated that they would be interested in using Charlie in CS1. Many responses were variants of "Yes", but students who responded Maybe (n=13) or No (n=23) typically explained their reasoning. Half (n=19) of these suggested that tools like Charlie would inhibit student learning. For instance, AQUALADYBUG noted, *'If I had questions on how to program a particular thing, using something like Charlie could help me clarify any questions I had by testing out different descriptions. But if I completely relied on something like Charlie as a tool in such a class, I feel like the whole point of me taking the class is overlooked and at some point becomes redundant."* Other students, including those who responded Yes, brought up how programmer skill level could play a role. TEALHERRING wrote, *"Yes, but I would want to maybe only try it out towards the end of the course, when I've already learned the process of coding and would like to see how an AI could work to streamline the process."* Other comments touched on academic integrity (*"I don't think so unless my teacher explicitly endorsed it because I'm terrified of plagiarism!"* - CRIMSONWORM).

More students supported using tools like Charlie in their own future programming practice (n=95). Maybe (n=20) and No (n=4) respondents again provided more explanation: two common themes included Charlie's limitations and usefulness for different kinds

Scale	Mean	Correlation with Success Rate (τ)
Ignorant - Knowledgeable	3.68	0.16*
Machinelike - Humanlike	2.39	0.12*
Responding rigidly - Responding elegantly	3.13	0.09
Unfriendly - Friendly	4.2	0.008
Incompetent - Competent	3.58	0.19*

Table 7: Mean student responses to Charlie perception questions (1=left endpoint, 5=right endpoint), adapted from Wang et al. [99], and correlation with success rate. * indicates statistical significance.

Question	Mean	Correlation with Success Rate (τ)
Charlie is capable of taking over complicated tasks.	3.24	0.03
Charlie might make sporadic errors.	2.15	0.18*
I was able to understand why things happened.	2.24	-0.34*
I can rely on Charlie.	2.95	-0.17*
Automated systems generally work well.	2.46	-0.14

Table 8: Mean student responses to Charlie trust questions (1 = Strongly agree; 5 = Strongly disagree), adapted from Körber [51], and correlation with success rate. * indicates statistical significance.

of problems: *"If Charlie improved, then it should be able to generate simple functions for me, in which I don't have to repeat myself"* (*purpleCarp*).

8.5 AI Attitudes

Students were asked whether they felt optimistic or pessimistic about AI's future impact on society. About two-thirds of students were optimistic; however, students pursuing a programming major (Computer Science, Data Science, or Media Arts and Science) were notably more optimistic than other students (80% optimistic compared to 63% of other majors). There was no difference in task performance between optimists and pessimists (pass@1 rate = 0.22 for both).

Students were also asked to compare the ethicality of Charlie with three other AI deployment scenarios. Most students found Charlie less ethically concerning in each comparison (Figure 9). Student responses to these questions did not differ reliably in relation to their success rate or pass rate.

9 DISCUSSION

In the previous sections we discussed our three main research questions – we summarize the findings together here:

- RQ1: We find that some students can effectively prompt a Code LLM, but it often takes numerous attempts. Students overall found the task mentally demanding. Prior experience and first-generation status are correlated, positively and negatively respectively, with success.
- RQ2: The most common issues students report relate to the Code LLM misunderstanding their descriptions and issues with generated code. Both students themselves and our analysis of the data suggest that the stochastic nature of the Code LLM may impact student experiences. We find limited differences between students regarding problem difficulty.
- RQ3: Students' most common mental model for the Code LLM was a data structure with keyword lookup. The most common prompting strategy that students developed was to expand their prompts, making them more detailed and more Pythonic. Students viewed the model as fairly capable and somewhat reliable. However, they expressed a range of opinions about whether Code LLMs would be appropriate for CS1.

In this section we draw connections between our findings and related work and discuss their broader implications.

9.1 The Natural-Language-to-Code Task is Challenging

The emergence of LLMs have led some to conclude that this is the "end of programming" [65, 100]. In contrast, we find that *beginners who can write code nevertheless struggle to write natural language prompts for LLMs*. We carefully select problems that are similar (or identical) to those they completed to pass CS1. The average participant solves 57% of the assigned problems, but only after several repeated attempts and with automatic feedback on code correctness. Our study contributes to the existing work on beginner interactions with Code LLMs by measuring how well students can use Code LLMs to solve problems at their own programming skill level, rather than in the context of a learning activity, where students may not be expected to able to write the code themselves. Despite the fact that all of our participants had passed CS1, which required writing code to solve problems like those in our study, many of them struggled to write natural language descriptions to lead a Code LLM to solve similar tasks.

On the whole, our findings reveal a somewhat higher level of difficulty in using Code LLMs than other studies [19, 48, 78], though it can be challenging to compare across diverse student populations, study designs, and problem types. Our results align most closely with those from Denny et al. [19]'s subsequent study of students with just two weeks of programming instruction. Although their study used only 3 problems and had less experienced programmers, they observed similar challenges: 86% of students eventually solved

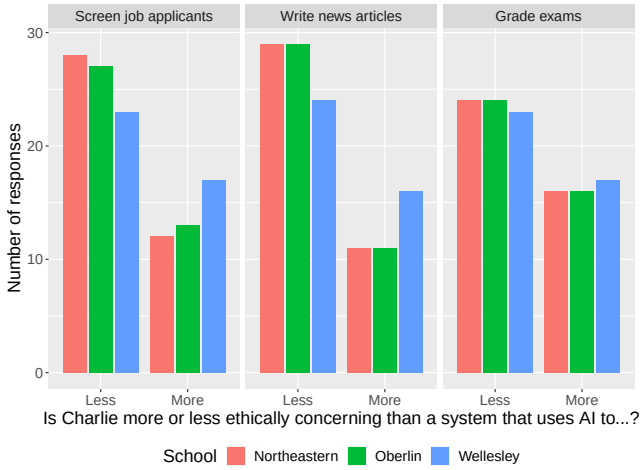


Figure 9: Student perceptions of Charlie's ethicality as compared to other AI scenarios

their easiest problem, but only 65% solved their hardest task. This is close to the average eventual success rate that we observe.

9.2 Not a Panacea for Non-Expert Programming

Learning an effective process for how to prompt a Code LLM is the key to interacting successfully with it in the long term. Existing work on experts reveals different "modes" of interaction [6]. Our findings suggest that unlike experts, near-novices *do not develop well-defined strategies for how to prompt.* Students added more detail to their previous prompts, even when it would have been better to start from scratch. In addition, students' prompting abilities did not observably improve during the study (§8.3). Students' failure to develop effective strategies may also be linked to their incorrect mental models of how Code LLMs work (§8.1). These results suggest that prompting, like most ways of interacting with code, needs to be explicitly taught to be used effectively.

Kazemitabaar et al. [48] present a study of pre-college students that suggests Code LLMs can improve learning outcomes. They compare student performance with and without access to the Code LLM, and provide considerable support to participants, such as instructor feedback and access to expert-written descriptions of the problem. In three of their task categories, both students with and without access to a Code LLM were able to complete 100% of the tasks, making it difficult to understand the contribution of the Code LLM. In the two more challenging categories, students benefited from the Code LLM, but they also relied heavily on the expert-written description (reusing it around 40% of the time). Together with our results, we take this to indicate that Code LLMs can be useful to beginners, but that writing prompts remains a barrier. This highlights the importance of understanding why Code LLMs and beginning programmers struggle to understand each other: Kazemitabaar et al. [48] argue that Code LLMs could positively impact student learning, but our results demonstrate a variety of ways that these interactions currently fail.

Our findings provide fine-grained evidence about student challenges that have implications for complete novices, as well as the beginners we study. The results in §8 highlight how effective prompting requires skills that complete novices do not possess. Figure 8 visualizes how students described their start and end approaches to editing, showing that many students who started out writing prompts as for a human transition into using more coding terminology by the end of the study. These participants picked up on a key property of Code LLMs: they are trained on expert-written code and documentation and expect natural language prompts to utilize coding terminology. The strategies that were most effective for our beginners would not be available to true novices.

9.3 Don't Assume a Mental Model of AI

Our study suggests that students have *incomplete mental models* of how Code LLMs work. Although participants knew they were interacting with an AI code generation tool and the majority (n=88, 73% in the post survey) had heard of GPT-3, Github Copilot, or Codex, when asked how they thought our system worked, only 19 students mentioned these models. A notable feature of responses was the number of detailed, but incorrect explanations. The majority of students who gave examples identified a keyword-based lookup strategy, like the dictionaries they had learned about in CS1.

These mental models fail to explain one aspect of Codex that students find frustrating: its stochastic responses. Students are familiar with errors that persist after editing their code. Code LLMs introduce a related but novel experience: submitting the same prompt and getting a different program (§7.3). This does not occur in standard CS1 settings and cannot be explained by the database/dictionary mental model of Code LLMs that most participants described. Without a well-developed understanding of why this happens, students have simply added another unknown computational behavior to their coding experience.

We note that although Prather et al. [78] report that several of their participants described models as having sentience or agency, none of our participants did. This may reflect the growing public awareness of generative AI between their study and ours, resulting in more realistic attitudes about the capabilities of large language models in our population. Our students seem to understand what AI models can do, but not how they do it.

9.4 Implications for Educators

Recent work has shown that Code LLMs can solve CS exams or homework assignments *given the educator's description of the problem* [17, 25]. Our findings show that although Code LLMs can solve CS1 problems, CS1 students cannot necessarily prompt Code LLMs to solve CS1 problems. Our findings reiterate the importance of key skills taught in CS1: code comprehension, problem decomposition, and the ability to describe computational problems clearly.

While we do not study learning outcomes explicitly, we find mixed support for Code LLMs as pedagogical tools. The survey portion of our experiment included questions about participants' attitudes towards Code LLMs. About two-thirds of participants expressed interest in using similar technology in CS1. Some participants mentioned that the task helped them remember Python concepts that they had forgotten, or even learn new features (such as

list comprehensions for Oberlin students). Others felt that it helped them practice describing technical tasks in natural language; Code LLMs could be used to provide feedback on Explain In Plain English (EiPE) questions [16, 64], which many educators see as valuable, but difficult to use without automation [28]. Recent work on students' perceptions of automatically-graded EiPE questions provides guidelines that may serve as a first step towards using Code LLMs as automatic backends [44].

On the other hand, a sizeable number of students did not support using Code LLMs in CS1. Some students expressed ethical concerns. Many questioned whether coming to rely on Code LLMs would diminish their knowledge of programming or their sense of fulfillment. Our survey data also highlights a key challenge of contemporary AI: explainability. Students gave Charlie higher ratings for capability than interpretability. Our findings here complement Sun et al. [93]'s exploration of Code LLM explainability needs identified by expert programmers, and Prather et al. [78]'s finding of students' "slow accept" mode, where students spent a lot of time reading code generated by Copilot and deciding whether or not to accept it.

By shedding light on how students feel about Code LLMs, our work augments Lau and Guo [52]'s investigation of CS educators' perspectives on Code LLMs. Our studies were conducted at a similar moment when Code LLMs had recently gained public prominence, but few educators or students had much experience with them. Our students and the educators in Lau and Guo [52] raise strikingly similar concerns about ethics and negative impacts on student learning. Denny et al. [19]'s subsequent experiment found similar concerns among currently enrolled CS1 students.

The large scale of our study also allows us to contribute data to the debate over equity in Lau and Guo [52]'s study, who show conflicting perspectives among educators: some felt that Code LLMs could strengthen the digital divide between students, while others felt that Code LLMs could improve diversity in CS. On the whole, our findings strengthens concerns. We show that students with extracurricular programming experience have an advantage, echoing Kazemitabaar et al. [48]'s finding that more experienced programmers benefit more from using Code LLMs. We also show that prompts written by first generation college students have reliably lower pass@1 rates. Educators should weigh the potential benefits of adopting this new technology against the possibility that it might exacerbate existing equity issues [41].

Finally, our students are ambivalent towards AI systems in general. Around two-thirds were optimistic about AI's impact on society in the future, similar to the proportion interested in using Charlie in CS1. This leaves a sizeable number of beginners who are concerned about AI or uninterested in its use in CS1. Our findings capture a nuanced portrait of how young adults perceive generative AI for programming, captured at a moment where generative AI was increasingly prominent in popular media.

9.5 Model Selection for Human-AI Interaction Research

One issue for studies such as ours is the rapid pace of research and development in machine learning. Running lab experiments with humans takes time. However, current proprietary models are often updated or deprecated with very little warning. This study used OpenAI's Codex, which provides state-of-the-art Code LLM performance but came with significant risks. In the middle of our study, OpenAI announced that Codex would be deprecated within a week, which would have seriously compromised our results; after much public concern, they eventually delayed the deprecation until early 2024.

The mismatch between the timescale of ML development and human-subjects research makes it difficult to complete studies using state-of-the-art models, which are largely proprietary. Based on our experience, we recommend not using proprietary models, although this may come with a trade-off in terms of performance, and imposes significant computational requirements for the research team (since alternatives require access to significant GPU resources). Nonetheless, we strongly suggest the use of open source models [59, 85] in future work, and potentially for classroom use, to avoid sudden loss of access. This is an example of an ongoing equity concern for researchers and educators.

9.6 Timeliness

Conducting work with non-experts and Code LLMs in early 2023 captures a specific moment in the evolution of this technology. Our participant pool represents students who mostly completed CS1 before Code LLMs became commonplace. Collecting this data now is paramount to our understanding of baseline interactions with Code LLMs for students without previous exposure. In the future, the controlled background knowledge of this study will become increasingly hard to come by, both at our institutions and farther afield.

We also see our work as timely because of the struggles and strategies, or lack thereof, that we identify. As computing resources become increasingly directed towards Code LLM technology [58], work such as ours has the potential to impact how companies develop their models, tutorials, and interfaces. We find that non-experts struggle to execute the full prompt and edit cycle, even with an interface that identifies output correctness. If this trend generalizes to other non-expert groups, Code LLM technology may strengthen the digital divide between expert and non-expert programmers, adding to the wide ranging list of ethical concerns about generative AI [9, 49, 61].

10 THREATS TO VALIDITY

A major challenge of studying human-AI interaction is that AI capabilities and popular awareness of them change quickly. ChatGPT was released between our pilot and main experiment; as a result, students' knowledge and experience with large language models underwent significant growth during our experiment. We observed a statistically significant improvement in task performance for students who took the study in the last month. This may spring from increased familiarity with large language models such as ChatGPT or from more recent exposure to CS1 material.

Although we recruited participants who had completed CS1 and no subsequent CS courses, their programming backgrounds were not homogeneous. Some participants had taken a prior programming course in high school or in college, and some were concurrently enrolled in a programming course. We study the effects of

additional programming experience in §6.3. In addition, since we recruited students who had taken CS1 as early as Fall 2021, some participants reported having forgotten programming concepts or terms in the intervening time.

Several factors may have biased participants towards reporting positive perceptions of our system. While we ensured that the experimenter running the study was not a educator at the participant's institution, participants were aware that the study involved one of their professors and may have responded more positively as a result. In addition, students may have answered questions about text-to-code more positively because of the anthropomorphic qualities of our system design; several commented about the appealing affect of the Charlie mascot in post-study questions. Charlie may have also had an effect on students' level of task perseverance [54]. Finally, novelty bias is always a potential concern when evaluating novel interfaces or systems, as-is self-selection bias for stand-alone studies.

11 CONCLUSION

We present results from a large-scale, multi-institution study of how near-novices interact with Code LLMs. Our novel experimental design allows us to isolate the prompt writing and editing tasks, by using a lab-based experiment in which participants write natural language descriptions of tasks and receive automated feedback on the correctness of generated code.

Our results suggest that students who have complete a single CS course find using Code LLMs challenging, even with tasks at an appropriate skill level. Our findings highlight the various barriers that they face, ranging from distilling their problem understanding into words, using coding terminology, understanding generated code, and grappling with the stochasticity of Code LLM output. We show that certain groups of students, most notably, first-generation college students, face additional difficulties, raising equity issues related to the deployment of Code LLMs in the classroom. We also illustrate how students' incorrect mental models of how Code LLMs operate inhibit their ability to develop effective prompting strategies. Moreover, our qualitative results provide insight into how beginning programmers feel about introducing Code LLMs in the classroom, bringing their voices into an key contemporary debate and complementing existing work on educators' perspectives.

Our findings suggest that Code LLMs do not signal the "end of programming": in fact, they highlight the many ways in which Code LLMs remain inaccessible to non-experts. We hope that our findings will motivate renewed effort towards democratizing programming by closing this gap.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Shriram Krishnamurthi for their thoughtful feedback. We thank our colleagues who helped us recruit participants and who provided CS1 problems that we adapted. We would also like to thank Rachelle Hu for her work on the Charlie platform prototype. This work is partially supported by the National Science Foundation (SES-2326173, SES-2326174, and SES-2326175). We thank Northeastern Research Computing and the New England Research Cloud for providing computing resources.

REFERENCES

[1] Andrea Agostinelli, Timo I. Denk, Zalán Borsos, Jesse Engel, Mauro Verzetti, Antoine Caillon, Qingqing Huang, Aren Jansen, Adam Roberts, Marco Tagliasacchi, Matt Sharifi, Neil Zeghidour, and Christian Frank. 2023. MusicLM: Generating Music From Text. http://arxiv.org/abs/2301.11325

[2] Nader Akoury, Shufan Wang, Josh Whiting, Stephen Hood, Nanyun Peng, and Mohit Iyyer. 2020. STORIUM: A Dataset and Evaluation Platform for Machine-in-the-Loop Story Generation. http://arxiv.org/abs/2010.01717 arXiv:2010.01717 [cs].

[3] 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. http://arxiv.org/abs/2306.04556

[4] Bruce W. Ballard and Alan W. Biermann. 1979. Programming in Natural Language: "NLC" as a Prototype. In *Annual Conference of the ACM*. Association for Computing Machinery, New York, NY, USA, 228–237. https://doi.org/10.1145/800177.810072

[5] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. http://arxiv.org/abs/2206.01335 arXiv:2206.01335 [cs].

[6] 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

[7] 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 (Jan. 2009), 71–81. https://doi.org/10.1007/s12369-008-0001-3

[8] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software* 67, 1 (2015), 1–48. https://doi.org/10.18637/jss.v067.i01

[9] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. ACM, Virtual Event Canada, 610–623. https://doi.org/10.1145/3442188.3445922

[10] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2023. Taking Flight with Copilot: Early Insights and Opportunities of AI-Powered Pair-Programming Tools. *Queue* 20, 6 (Jan. 2023), 35–57. https://doi.org/10.1145/3582083 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[11] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering* 49, 7 (July 2023), 3675–3691. https://doi.org/10.1109/TSE.2023.3267446

[12] Le Chen, Xianzhong Ding, Murali Emani, Tristan Vanderbruggen, Pei-Hung Lin, and Chunhua Liao. 2023. Data Race Detection Using Large Language Models. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. ACM, Denver CO USA, 215–223. https://doi.org/10.1145/3624062.3624088

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. http://arxiv.org/abs/2107.03374 arXiv:2107.03374 [cs].

[14] CodeWhisperer. 2023. ML-powered Coding Companion – Amazon CodeWhisperer – Amazon Web Services. https://aws.amazon.com/codewhisperer/

[15] Github Copilot. 2023. Github Copilot Your AI pair programmer. https://github.com/features/copilot

[16] 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*. ACM, Atlanta Georgia USA, 591–596. https://doi.org/10.1145/2538862.2538911

[17] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, Zhen Ming, and Jiang. 2022. GitHub Copilot AI pair programmer: Asset or Liability? https://doi.org/10.48550/ARXIV.2206.15331

[18] Maria De-Arteaga, Riccardo Fogliato, and Alexandra Chouldechova. 2020. A Case for Humans-in-the-Loop: Decisions in the Presence of Erroneous Algorithmic Scores. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376638

[19] 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. http://arxiv.org/abs/2307.16364 arXiv:2307.16364 [cs].

[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*. ACM, Athens Greece, 49–61. https://doi.org/10.1145/3459990.3460712

[21] Stephen H. Edwards. 2004. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. ACM, Norfolk Virginia USA, 26–30. https://doi.org/10.1145/971300.971312

[22] 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.

[23] Kasra Ferdowsi, Ruanqianqian Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. 2023. Live Exploration of AI-Generated Programs. http://arxiv.org/abs/2306.09541 arXiv:2306.09541 [cs].

[24] Sally Fincher, Raymond Lister, Tony Clear, Anthony Robins, Josh Tenenberg, and Marian Petre. 2005. Multi-institutional, multi-national studies in CSEd Research: some design considerations and trade-offs. In *Proceedings of the first international workshop on Computing education research (ICER '05)*. Association for Computing Machinery, New York, NY, USA, 111–121. https://doi.org/10.1145/1089786.1089797

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

[26] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco CA USA, 1229–1241. https://doi.org/10.1145/3611643.3616243

[27] Matthew Flatt, Matthias Felleisen, Robert Bruce Findler, and Shriram Krishnamurthi. 2001. *How To Design Programs*. MIT Press.

[28] 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*. ACM, Virtual Event USA, 1163–1169. https://doi.org/10.1145/3408877.3432539

[29] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. http://arxiv.org/abs/2204.05999 arXiv:2204.05999 [cs].

[30] Marwa Gadala. 2017. Automation bias: exploring causal mechanisms and potential mitigation strategies. https://api.semanticscholar.org/CorpusID:41123263

[31] Chuqin Geng, Haolin Ye, Yixuan Li, Tianyu Han, Brigitte Pientka, and Xujie Si. 2022. Novice Type Error Diagnosis with Natural Language Models. http://arxiv.org/abs/2210.03682 arXiv:2210.03682 [cs].

[32] Kate Goddard, Abdul V. Roudsari, and Jeremy C. Wyatt. 2012. Automation bias: a systematic review of frequency, effect mediators, and mitigators. *Journal of the American Medical Informatics Association : JAMIA* 19 1 (2012), 121–7.

[33] Jamie Gorson and Eleanor O'Rourke. 2020. Why do CS1 Students Think They're Bad at Programming?: Investigating Self-efficacy and Self-assessments at Three Universities. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. ACM, Virtual Event New Zealand, 170–181. https://doi.org/10.1145/3372782.3406273

[34] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. Publisher: ACM New York, NY, USA.

[35] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, and others. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. Publisher: Now Publishers, Inc..

[36] 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

[37] 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.

[38] George E. Heidorn. 1974. English as a Very High Level Language for Simulation Programming. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/800233.807050

[39] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Zurich, Switzerland, 837–847.

[40] Jack Hollingsworth. 1960. Automatic graders for programming classes. *Commun. ACM* 3, 10 (1960), 528–529. Publisher: ACM New York, NY, USA.

[41] Kenneth Holstein and Shayan Doroudi. 2021. Equity and Artificial Intelligence in Education: Will "AIEd" Amplify or Alleviate Inequities in Education? http://arxiv.org/abs/2104.12920 arXiv:2104.12920 [cs].

[42] Kenneth Holstein, Bruce M McLaren, and Vincent Aleven. 2018. Student learning benefits of a mixed-reality teacher awareness tool in AI-enhanced classrooms. In *Artificial Intelligence in Education: 19th International Conference, AIED 2018, London, UK, June 27–30, 2018, Proceedings, Part I 19*. Springer, 154–168.

[43] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rygGQyrFvH

[44] Silas Hsu, Tiffany Wenting Li, Zhilin Zhang, Max Fowler, Craig Zilles, and Karrie Karahalios. 2021. Attitudes Surrounding an Imperfect AI Autograder. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–15. https://doi.org/10.1145/3411764.3445424

[45] Daphne Ippolito, Ann Yuan, Andy Coenen, and Sehmon Burnam. 2022. Creative Writing with an AI-Powered Writing Assistant: Perspectives from Professional Writers. http://arxiv.org/abs/2211.05030 arXiv:2211.05030 [cs].

[46] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[47] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. *Proceedings of the AAAI Conference on Artificial Intelligence* 37, 4 (June 2023), 5131–5140. https://doi.org/10.1609/aaai.v37i4.25642

[48] 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

[49] Heidy Khlaaf, Pamela Mishkin, Joshua Achiam, Gretchen Krueger, and Miles Brundage. 2022. A Hazard Analysis Framework for Code Synthesis Large Language Models. http://arxiv.org/abs/2207.14157 arXiv:2207.14157 [cs].

[50] 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, Rome, Italy, 199–206. https://doi.org/10.1109/VLHCC.2004.47

[51] Moritz Körber. 2018. Theoretical considerations and development of a questionnaire to measure trust in automation. In *Congress of the International Ergonomics Association*. Springer, 13–30.

[52] 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 (ICER '23)*. Association for Computing Machinery, New York, NY, USA, 106–121. https://doi.org/10.1145/3568813.3600138

[53] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (Oct. 2009), 65–65. https://doi.org/10.1609/aimag.v30i4.2262

[54] 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.

[55] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing Code Explanations Created by Students and Large Language Models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. ACM, Turku Finland, 124–130. https://doi.org/10.1145/3587102.3588785

[56] 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*. ACM, Toronto ON Canada, 563–569. https://doi.org/10.1145/3545945.3569770

[57] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[58] Jonathan Vanian Leswing, Kif. 2023. ChatGPT and generative AI are booming, but the costs can be extraordinary. https://www.cnbc.com/2023/03/13/chatgpt-and-generative-ai-are-booming-but-at-a-very-expensive-price.html

[59] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! http://arxiv.org/abs/2305.06161 arXiv:2305.06161 [cs].

[60] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2023. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. http://arxiv.org/abs/2303.17125 arXiv:2303.17125 [cs].

[61] Q. Vera Liao and Jennifer Wortman Vaughan. 2023. AI Transparency in the Age of LLMs: A Human-Centered Research Roadmap. http://arxiv.org/abs/2306.01941 arXiv:2306.01941 [cs].

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

[63] Vivian Liu, Tao Long, Nathan Raw, and Lydia Chilton. 2023. Generative Disco: Text-to-Video Generation for Music Visualization. http://arxiv.org/abs/2304.08551

[64] 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

[65] Farhad Manjoo. 2023. It's the End of Computer Programming as We Know It. (And I Feel Fine.). *The New York Times* (June 2023). https://www.nytimes.com/2023/06/02/opinion/ai-coding.html

[66] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (Nov. 2019), 1–23. https://doi.org/10.1145/3359174

[67] L. A.. Miller. 1981. Natural language programming: styles, strategies, and contrasts. *IBM Systems Journal* 20, 2 (June 1981), 184–215. https://doi.org/10.1147/sj.202.0184

[68] Piotr Mirowski, Kory W. Mathewson, Jaylen Pittman, and Richard Evans. 2023. Co-Writing Screenplays and Theatre Scripts with Language Models: Evaluation by Industry Professionals. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–34. https://doi.org/10.1145/3544548.3581225

[69] 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].

[70] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52. https://doi.org/10.1109/MC.2016.200

[71] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. http://arxiv.org/abs/2307.08177 arXiv:2307.08177 [cs].

[72] National Center for Women & Information Technology. 2023. NCWIT Guide to Demographic Survey Questions. https://docs.google.com/document/d/1E_CSANwOqbKjEG27woNbGZ09JIXUfAf4Cp9j8g5DFak/

[73] 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. https://doi.org/10.48550/ARXIV.2203.13474

[74] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, C. J. Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2023. GPT-4 Technical Report. http://arxiv.org/abs/2303.08774 arXiv:2303.08774 [cs].

[75] David Lorge Parnas and Jan Madey. 1995. Functional documents for computer systems. *Science of Computer Programming* 25, 1 (Oct. 1995), 41–61. https://doi.org/10.1016/0167-6423(95)96871-J

[76] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. http://arxiv.org/abs/2302.06590 arXiv:2302.06590 [cs].

[77] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. http://arxiv.org/abs/2302.04662 arXiv:2302.04662 [cs].

[78] 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

[79] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, and others. 2019. Language models are unsupervised multitask learners. *OpenAI* 1, 8 (2019), 9. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[80] Iulian Radu and Bertrand Schneider. 2019. What can we learn from augmented reality (AR)? Benefits and drawbacks of AR for inquiry-based learning of physics.

In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–12.

[81] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-Shot Text-to-Image Generation. In *Proceedings of the 38th International Conference on Machine Learning*. PMLR, 8821–8831. https://proceedings.mlr.press/v139/ramesh21a.html ISSN: 2640-3498.

[82] Hemilis Joyse Barbosa Rocha, Patrícia Cabral De Azevedo Restelli Tedesco, and Evandro De Barros Costa. 2023. On the use of feedback in learning computer programming by novices: a systematic literature mapping. *Informatics in Education* 22, 2 (2023), 209. Publisher: Institute of Mathematics and Informatics.

[83] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-Resolution Image Synthesis with Latent Diffusion Models. IEEE Computer Society, 10674–10685. https://doi.org/10.1109/CVPR52688.2022.01042

[84] 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 (IUI '23)*. Association for Computing Machinery, New York, NY, USA, 491–514. https://doi.org/10.1145/3581641.3584037

[85] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. http://arxiv.org/abs/2308.12950 arXiv:2308.12950 [cs].

[86] Jean E. Sammet. 1966. The Use of English as a Programming Language. *Commun. ACM* 9, 3 (March 1966), 228–230. https://doi.org/10.1145/365230.365274

[87] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive Test Generation Using a Large Language Model. https://doi.org/10.48550/arXiv.2302.06527 Issue: arXiv:2302.06527 _eprint: 2302.06527.

[88] David E Shaw, William R Swartout, and C Cordell Green. 1975. Inferring LISP Programs From Examples. In *IJCAI*, Vol. 75. 260–267.

[89] 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), 3511599. https://doi.org/10.1145/3511599

[90] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 15–26.

[91] Linda J Skita, Kathleen Mosier, and Mark D. Burdick. 2000. Accountability and automation bias. *International Journal of Human-Computer Studies* 52, 4 (2000), 701–717. https://doi.org/10.1006/ijhc.1999.0349

[92] Katta Spiel, Oliver L. Haimson, and Danielle Lottridge. 2019. How to do better with gender on surveys: a guide for HCI researchers. *Interactions* 26, 4 (June 2019), 62–65. https://doi.org/10.1145/3338283

[93] 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*. ACM, Helsinki Finland, 212–228. https://doi.org/10.1145/3490099.3511119

[94] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. 2017. Tracediff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 107–115.

[95] TabNine. 2023. AI Assistant for Software Developers | Tabnine. https://www.tabnine.com/

[96] Priyan Vaithilingam, Elena L. Glassman, Peter Groenwegen, Sumit Gulwani, Austin Z. Henley, Rohan Malpani, David Pugh, Arjun Radhakrishna, Gustavo Soares, Joey Wang, and Aaron Yim. 2023. Towards More Effective AI-Assisted Programming: A Systematic Design Exploration to Improve Visual Studio IntelliCode's User Experience. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[97] 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

[98] Maarten Van Mechelen, Rachel Charlotte Smith, Marie-Monique Schaper, Mariana Tamashiro, Karl-Emil Bilstrup, Mille Lunding, Marianne Graves Petersen, and Ole Sejer Iversen. 2023. Emerging technologies in K–12 education: A future HCI research agenda. *ACM Transactions on Computer-Human Interaction* 30, 3 (2023), 1–40. Publisher: ACM New York, NY.

[99] Qiaosi Wang, Koustuv Saha, Eric Gregori, David Joyner, and Ashok Goel. 2021. Towards Mutual Theory of Mind in Human-AI Interaction: How Language Reflects What Students Perceive About a Virtual Teaching Assistant. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–14. https://doi.org/10.1145/3411764.3445645

[100] Matt Welsh. 2022. The End of Programming. *Commun. ACM* 66, 1 (Dec. 2022), 34–35. https://doi.org/10.1145/3570220 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[101] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. http://arxiv.org/abs/2308.04748 arXiv:2308.04748 [cs].

[102] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, San Diego CA USA, 1–10. https://doi.org/10.1145/3520312.3534862

[103] J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–21. https://doi.org/10.1145/3544548.3581388

[104] 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/ Publication Title: The GitHub Blog.

[105] 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 Study Design

A.1.1 Pilot Study. In late 2022, we ran an IRB-approved pilot study with 19 participants from all three institutions. These students had completed CS1 and at least one additional course, so they were ineligible for the main study. Overall, we made few changes after the pilot. The most consequential were to add an additional 15 minutes (75 minutes total) to the study window, increase participant compensation, and implement word wrapping in the interface to prevent excessive scrolling.

A.1.2 Problem Adaptation. Our problems were based on CS1 problems used at each of our three institutions. In most cases, we made small adaptations to the problems, both to make it less likely for students to recognize the exact problem, and to fit the constraints of the Code LLM task (i.e, changing printed output to returned output, avoiding library imports).

Figure 10 presents two examples of how we adapted problems. Figure 10a shows the original presentation of the problem that was adapted into *mod_end*. We added an additional parameter so that the function substitutes a given string for the 's' at the end of each string in the list. We also renamed the function. Note that in the original class setting, the problem was presented with three input/output pairs, as in our experimental design.

Figure 10b shows the original presentation of the problem that was turned into *find_multiples*. We changed the function to return the list of multiples rather than the number of multiples. As in our experiment, the original problem description contained three input/output pairs.

A.1.3 Problem Validation. By selecting from existing problems in the CS1 curricula, we ensured that the problems were at an appropriate difficulty level for our student population. In order to focus specifically on the human-model interaction, we also needed to ensure that the problems were an appropriate difficulty level for the code generation model: the model is capable of generating a solution, but only when it is appropriately prompted.

Because code generation models memorize common associations between function names and function bodies, it is important to ensure that the model cannot generate a passing implementation from the function name alone. We produced Codex generations from just the function signature for every problem, without any natural language prompt, and measured mean pass@1 rate. We renamed any functions with high pass@1 rates. For our final set of problems, the overall mean pass@1 for function signatures alone is 0.0519. The maximum pass@1 is 0.925, for the problem `exp`. This means that students generally need to provide a description of the function's intended behavior in order for the model to produce a correct implementation.

We also ensured that there was a prompt that would lead to a correct implementation for every problem. Each problem has an "expert" prompt written by one of the authors for which Codex produces a correct implementation. These prompts were not otherwise used as part of the experiment.

A.1.4 Test Case Validation. We rely on unit tests to check the correctness of model-generated code. These tests also produce feedback for students about the model's generated code. We built an initial suite of test cases for each problem by taking tests from grading rubrics and other class resources. We used test coverage and mutation testing [46] to identify missing test cases and build more robust test coverage, while keeping the number of test cases per problem to a size that can be easily displayed.

A.2 Qualitative Analysis

As described in the main body of the paper, the analysis of the qualitative data was done by two researchers with previous qualitative analysis experience. The aim was to identify common themes in the data set, rather than build a generalizable theory. Below we outline the analyses performed on three types of data: (1) data about student experience and demographics, (2) free-response questions about future use of Charlie, and (3) the semi-structured interview responses. We provide the full codebooks, with definitions, for all data types as part of our Supplemental Materials at https://doi.org/10.17605/OSF.IO/V2C4T.

A.2.1 Student Experience & Demographics. We used thematic analysis for the post survey questions, beginning with the Language, Major, and Experience questions. Codes were developed inductively - the two researchers independently developed codes and then iterated on a code set via conversation and consensus. We did not calculate inter-rated reliability for these questions, as their specific use was for quantitative analysis rather than for specific qualitative trends [66]. Once the researchers arrived at a tentative codebook they independently coded and iterated until there was complete consensus on all codes for all data points as part of the post survey. This took one round to normalize code application (e.g., Computer Science was not coded as a Natural Science) and then a second round where the codes were complete, but typos were identified.

A.2.2 Free Response Questions. These questions (UseCharlie and Foresee) were coded second out of the three kinds of qualitative data. This process initially followed a similar inductive style to that described above. Due to the open-ended nature of these responses, both researchers then developed independent definitions for each code to provide clearer guidelines for inclusion/exclusion. They then met to merge their definitions and discuss any discrepancies. For instance, normalizing most definitions to start with "Mentions" and combining definitions or picking the more detailed. Then the researchers independently coded according to the consensus definitions. Arriving at consensus took two rounds. Two sets of codes were combined (two subcodes of Skill Level and two subcodes of Problem Difficulty) and Documentation/Code Understanding was re-coded due to clarifications in their definitions. The final round of coding identified only typos and unintentional omissions. Again, consensus was reached and inter-rated reliability was not calculated for these codes.

A.2.3 Semi-Structured Interview Analysis. We took a different approach to coding the semi-structured interview data than the post-survey data, as the responses varied significantly in length and precision. The details of the codebook development are described below, but the following process was conducted for all 8 questions:

3. mapPluralize

Write a function called `mapPluralize(mylist)` that takes a list of words and returns a new list with the plurals of each of the words.

The nouns are pluralized by adding the suffix `s`, e.g. the plural of `bagel` is `bagels`. You do not have to handle special cases like `kiss` -> `kisses`.

Examples:

```
print (mapPluralize(['assignment']))
# Results: ['assignments']

print (mapPluralize(['donut', 'muffin', 'bagel']))
# Results: ['donuts', 'muffins', 'bagels']

print (mapPluralize(['tree' ,'witch' , 'kiss', 'moose', 'alpaca']))
# Results: ['trees', 'witchs', 'kisss', 'mooses', 'alpacas']
```

(a) Study problem called `mod_end`

`findMultiplesOf` is a None function that prints all the multiples of a given factor between the given start and stop numbers, inclusive.

```
>>> findMultiplesOf(1,10,15)
 There are 0 multiples of 15 between 1 and 10.

>>> findMultiplesOf(1,100,17)
 17
 34
 51
 68
 85
 There are 5 multiples of 17 between 1 and 100.

>>> findMultiplesOf(1000,2000,177)
 1062
 1239
 1416
 1593
 1770
 1947
 There are 6 multiples of 177 between 1000 and 2000.
```

(b) Study problem called `find_multiples`

Figure 10: Original problem presentations

(1) The two coding researchers independently developed codes for a set of 15 non-overlapping interviews. They met to discuss their codes and general themes.
(2) They then coded 15 shared interviews to test the codes, add additional codes, and finalize definitions. They reached consensus on the codes and their application to the shared 15 interviews.
(3) To confirm their understanding of the codebook, they then coded 20 shared interviews and calculated percent agreement. Any codes with low agreement were discussed, had their definitions changed/edited, or were removed. The researchers then came to consensus on how to apply the codes to these data.
(4) The researchers then divided the remaining 70 interviews and coded independently, making use of a fixed codebook. The researchers did not code interviews they conducted themselves. They also independently recoded their original 15 datapoints.

The two researchers began by coding the last four interview questions, which they deemed more concrete. This process was primarily inductive. Computing percent agreement across the data, 87% of our 70 codes exhibited 90% or higher agreement (i.e. disagreement on 2 or less datapoints). 9 codes were less, with the minimum agreement being 75%.

The researchers then moved on to the first four interview questions – the last analysis performed on the qualitative data. This process was more deductive than previous analyses. For example, for Hardest/Easiest, the topic categories of problems (e.g. Loops, Conditionals) were particularly relevant to our analyses and the data suggest those categories as codes. The first two problems had the most variation in student responses – we attribute this to students' lack of knowledge of their process, as they found this task difficult overall. Therefore, the researchers focused on codes that err on the side of temporal attributes. Percent agreement was again calculated for these codes - 80% had 90% agreement or higher. Only two codes had agreement lower than 75% - the researchers discussed these codes significantly, reaching agreement on the generality of one code (Add Detail) and removing another code entirely.

B ADDITIONAL QUANTITATIVE RESULTS

B.1 Participant Demographics

In order to protect participant anonymity, we report responses to the open-ended demographics questions only if an identical response was submitted by at least 5 participants. Gender and race responses are shown in Table 9 and Table 10. The majority of responses to the question about ethnicity were unique. Due to the need to protect participant anonymity, we have chosen not to report this data.

Self-Reported Gender	N	Mean pass@1
Female	72	0.22
Male	30	0.21
Nonbinary	5	0.28
All other responses	13	0.24

Table 9: Self-reported gender of participants, capitalization normalized to title case.

Self-Reported Race	N	Mean pass@1
Asian	38	0.22
Black	6	0.11
East Asian	6	0.34
White	36	0.21
All other responses	34	0.23

Table 10: Self-reported race of participants, capitalization normalized to title case.

B.1.1 Statistical Analysis. We used Welch Two Sample t-tests to explore whether there were statistically reliable differences in pass@1 rates for students with different backgrounds. Table 11 shows the results.

B.2 Problem Difficulty

B.2.1 Statistical Analysis of Category Difficulty. A binomial mixed-effects model (Table 12) was fitted to prompt success as a binary

Group 1	Group 2	t	p
Domestic student	International student	-0.4	0.68
First generation college student	Not first generation college student	2.1	**0.04**
English in childhood household	No English in childhood household	1.02	0.31
Public high school	Private high school	0.1	0.92
Coding experience outside of CS1	No other coding experience	2.47	**0.02**
More than 1 math course	1 college math course	0.8	0.43

Table 11: Welch Two Sample t-tests to explore differences in pass@1 rates between demographic groups

Fixed effects	$\widehat{\beta}$	z	p
(Intercept)	-0.80 (+/- 0.6)	-1.35	0.18
Dictionaries	-0.70 (+/- 0.8)	-0.8	0.41
Lists	-0.55 (+/- 0.8)	-0.7	0.51
Loops	-0.92 (+/- 0.8)	-1.1	0.28
Math	-1.10 (+/- 0.8)	-1.3	0.19
Nested	0.83 (+/- 0.8)	1.0	0.32
Sorting	-1.75 (+/- 0.9)	-2.0	**0.045**
Strings	0.14 (+/- 0.8)	0.2	0.87
Wellesley	0.14 (+/- 0.4)	0.4	0.71
Oberlin	0.24 (+/- 0.4)	0.6	0.53
Dictionaries:Wellesley	-0.04 (+/- 0.6)	-0.1	0.95
Lists:Wellesley	0.37 (+/- 0.5)	0.7	0.50
Loops:Wellesley	0.62 (+/- 0.5)	1.2	0.25
Math:Wellesley	0.18 (+/- 0.5)	0.3	0.73
Nested:Wellesley	-0.98 (+/- 0.5)	-1.9	0.062
Sorting:Wellesley	-0.16 (+/- 0.6)	-0.3	0.77
Strings:Wellesley	-0.17 (+/- 0.5)	-0.4	0.73
Dictionaries:Oberlin	-0.52 (+/- 0.6)	-0.9	0.37
Lists:Oberlin	-0.05 (+/- 0.5)	-0.1	0.93
Loops:Oberlin	-0.33 (+/- 0.6)	-0.6	0.57
Math:Oberlin	0.14 (+/- 0.5)	0.3	0.79
Nested:Oberlin	-0.57 (+/- 0.5)	-1.1	0.28
Sorting:Oberlin	-0.10 (+/- 0.6)	-0.2	0.86
Strings:Oberlin	0.10 (+/- 0.5)	0.2	0.84

Table 12: Full results of binomial mixed-effects model fitted to problem category and institution.

outcome (1 if the prompt succeeded; 0 otherwise). The model included fixed effects of problem category, institution, and their interaction, and random effects of participant and problem. Treatment coding was used for institution, with Northeastern as the baseline category; deviation coding was used for category, since we were interested in whether any one category differed from the average problem difficulty.

B.2.2 Least-Solved Problems. To understand where struggles arise, we manually examined student responses to two problems: *laugh*, which has one of the lowest number of student successes, and *total_bill*, which has a mid-range success rate.

A challenging problem: `laugh`. One of the least-solved problems in our study was `laugh`. The intended function takes a number n and produces a string of n "ha"s, where the initial "ha" has n "a"s, and each subsequent laugh has one fewer "a".

Only two students were able to eventually succeed at this task (*orchidWalleye* and *magentaWeasel*). However, a manual inspection of all initial student descriptions reveals only one serious misunderstanding of the task (*tealPossum*) – see Table 13 for all students' initial descriptions.

A mid-range problem: `total_bill`. The task in `total_bill` is to compute the total of a grocery bill, using a list of grocery items and a sales tax rate. Each grocery item is itself a list containing the name of the item, a quantity, and a price. One expert description that reliably generates a working program is *Returns the sum of multiplying the second and third indices of each list in grocery_list, multiplied by 1 + sales_tax. Round to 2 digits.*

We manually inspect all descriptions for this problem. Of the 20 students who attempted this problem, 12 eventually succeed. All of these students follow a similar path: their first attempt omits the rounding step, leading one of the tests to fail. A handful of students also omit or incorrectly describe the sales tax step initially.

What about the students who never succeed? One student initially misunderstands the task, writing: *This function takes in a list of the item purchased, the price, the tax, and the overall sales tax. All of the prices and tax within the lists are added together. The sales tax is then multiplied by the outcome of the added prices, and then the result of the multiplication is added onto the total price. The total price is then returned as the output.* (*limeSalamander*)

The student has misunderstood a key detail in the structure of the lists: the two numbers are the quantity and price, so they should be multiplied, not added. Consequently, this prompt fails. However, their third description is accurate: *This function takes in a list of the item purchased, the amount of the item purchased, the price for each item, and the overall sales tax. The amount purchased is multiplied with the price for each item, creating a total amount. The sales tax is then multiplied by the outcome of the total amount, and then the result of the multiplication is added onto the total price. The total price is then returned as the output.*

Although the student initially misunderstood part of the problem, they are able to reread the input/output pairs and/or code, arriving at the correct interpretation eventually. However, their description still fails. This participant eventually runs out of time. The rest of the participants who never succeed submit accurate descriptions that omit key details, such as how to calculate the sales tax (6 participants) or the list positions of the price and quantity (5 participants).

Overall, the student prompts for `total_bill` demonstrate more issues in describing the problem than in understanding it. Although one participant misunderstands the task initially, they were able to quickly self-correct.

Participant	Initial Description	N submissions
aquaLadybug	If n is the input value, returns a combination of n strings, where each of the n strings consists of "h" followed by n occurrences of "a", and there is " " before each "h" except the first "h"	18
greenMoth	a function have initial input as 'ha' when input of size(int) is 1, when size+= 1 from 1, 'ha' will gain one more 'a'	2
orchidBeetle	Based on the inputted number, will return a laugh size where the number of "a"'s starts with the initial size, then decreases by one for each additional laugh.	4
tealPossum	return the number of words in a string	2
pinkFisher	the function laugh will take the input of an int and should output a string with the ha as many times as the input but also the number of a's is based on the number it is currently working with	4
magentaWeasel	Write a function which takes an integer size as an input, and uses a for loop to print an h followed by size a's and then a space, and then an h followed by size-1 a's and then a space, etc. until it prints a h followed by one a	7
aquamarineShrew	This function prints an 'h' and adds the corresponding amount of a's as the value provided. It then adds a space to the output. It subtracts 1 from the value and prints another h with less a's and repeats until the value of the number is 0	26
orchidWalleye	function adds 'a' to every 'h' based on input and will lower amount of 'a' until it reaches only 1 'a' after the 'h'	3
khakiBee	take in a number and write the word 'ha' but with as many 'a's as the number	7
pinkPerch	Produce a string, with each word starting with h and then however many a's the input says. Decrease the count of a's by one following the h for each word after.	5
orchidFlounder	the input generates a string where the number corresponds to how many items are in the string. each item in the string also starts with the letter 'h' and the letter 'a' is added to the letter 'h' based on the number of the input. However, only the first item in the string has the number of 'a' equal to the input, the following 'a' are added to 'h' by subtracting 1 from the input.	1
beigeBass	the code increases the number of the letters in "ha," depending on the input in an increasing factorial way	1
tomatoFisher	This function takes an integer and an input produces the word "ha" that number of times but the number of times "a" appears in each "ha" decreases by one until "ha"	2
crimsonVole	Takes in an integer 'n' input and outputs a string with 'n' words, 'h' as the first letter for each word, and 'n' number of 'a's after it, followed by 'h' as the first letter of the next word and 'n-1' number of 'a's after it and so on until we reach n = 1	1
lavenderPossum	Given an integer, return a string in the form 'ha' where the integer determines the number of a's and repeat the same pattern until there is one a	5
lavenderBat	The input takes in a number, say n, and produces a string that has n words. the first word is formed of one "h" and n number of "a". The number of "a" decreases by one for each next word	8
magentaDolphin	This function returns the number of laughs in a string, where a laugh is the character 'h' followed by any number of the character 'a'	9
linenBobcat	Counts the number of laughs, beginning with the given number of "a"s within it and descending by each laugh, totaling the given number of laughs.	2
grayVole	Takes size and uses recursion to produce that number of "ha" laughs with one less "a" with each "ha" until there is only one "a" left	8
thistleTrout	Using the given number, add that number of "a"s after an "h". Count down the number by 1, and add that number of "a"s after another "h" and repeat.	5

Table 13: Initial descriptions of the `laugh` problem from all 20 students who encountered it. *N submissions* describes how many times the specific student attempted `laugh` before succeeding or giving up.