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

ANONYMOUS AUTHOR(S)

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.

1 INTRODUCTION

Computer scientists have been working towards programming in natural language for decades [4, 24, 61], 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 [56–58], creative writing [2, 28, 48], and digital music [1, 44], 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 [49, 53, 71]. However, to fulfill their potential of democratizing these fields, models must be usable without extensive technical training at each stage of creation: 1) writing 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 [52]. 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 [32] and classical AI systems [35].

2023. Manuscript submitted to ACM

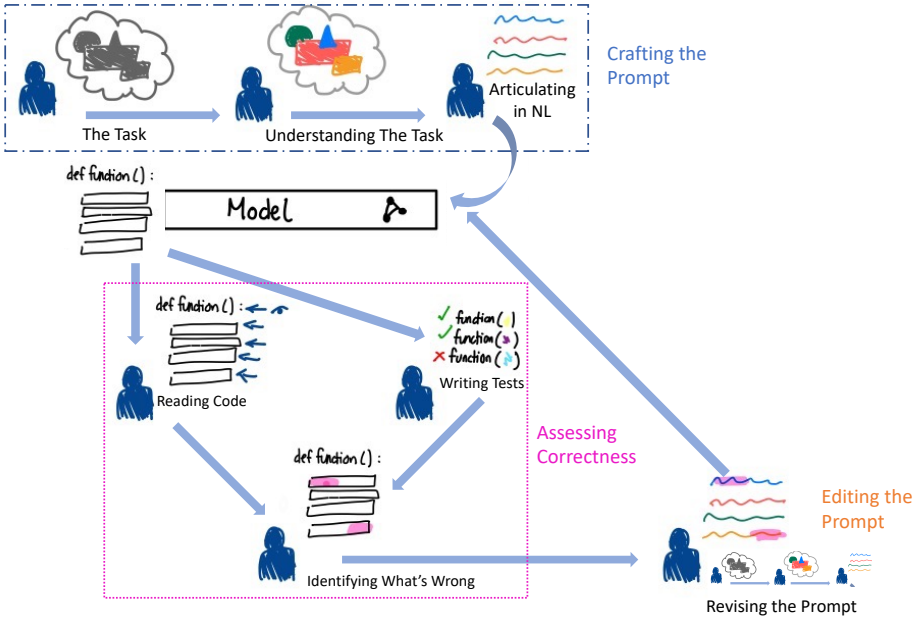


Fig. 1. Visualization of the multi-step process of querying an large language mode of code (Code LLM). The user starts with crafting their prompt in natural language (NL). They provide the prompt to the model, which produces code. The user then assesses the correctness of the generated code. If there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

There is a growing body of work studying how non-experts programmers use AI-assisted programming systems in naturalistic settings [55]. 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.

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 [46, 67]. 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.

2 RELATED WORK

Our study contributes to the growing body of work on interactions with automated code generation technology, specifically large language models of code (Code LLMs).

Experienced programmers and LLMs. Although Github Copilot and other Code LLMs impact beginning programmers – the subjects of our paper – they are typically promoted as productivity-boosting technology for experienced programmers. Recent in-the-wild studies and surveys indicate that LLM-powered autocomplete is popular with expert programmers, improves their self-perception of productivity, and shifts their work from writing code to understanding LLM-generated code [10, 41, 49]. In contrast, our study of beginning programmers reveals they have mixed success with writing natural language prompts and they exhibit significant struggles in understanding LLM-generated code.

Grounded Copilot finds that experienced programmers prefer prompting Copilot with natural language comments over code, and write more comments with Copilot [6, Section 4.2.3]. We study beginners and focus on the natural language to code task. Our larger population (120 vs. 20) and diverse set of tasks (48 vs. 4) gives us the power needed to perform statistical analysis.

Vaithilingam et al. [65] present the earliest academic study of Copilot with 24 students (undergraduate, masters, and PhD students) and three tasks. Their main finding is that although participants enjoyed using Copilot, it did not improve correctness or completion times. They report that participants often struggled to validate Copilot-written code, and to correct it when it is wrong. In our experiment, we automate the testing step, which is particularly hard for beginners, allowing us to focus on how participants write and edit natural language prompts.

Non-experts and LLMs. Like us, numerous researchers have considered the impact of using Code LLM for the code-to-text task with non-experts, specifically in educational settings. Kazemitabaar et al. [30], Prather et al. [55], and Denny

et al. [15] study student interaction directly, whereas Lau and Guo [34] research how educators view the role of Code LLMs in their classrooms.

CodingSteps is a web-based Python learning environment that allows users to query Codex [30]. The paper studies 69 participants (10–17 years old) without text-based programming experience and find that students with access to Codex do better at programming tasks, with no negative impact on their ability to program without Codex later. The study design presents students with teacher-written problem descriptions; as a result, 32% of student prompts are identical to the assignment description. Our experiment is designed explicitly to avoid this: we do not show participants any natural language task descriptions, but rather describe the problems via input/output pairs.

Prather et al. [55] had 19 students work on a homework problem using Copilot towards the end of a CS1 class. They find that novices struggle to use Copilot, and our paper reaches similar conclusions on even simpler problems. Promptly [15] studies 54 students writing prompts for three CS1 problems. The substantially larger scale of our study (120 students and 48 problems) allows us to explore trends in more depth.

Lau and Guo [34] interview 20 CS1/CS2 instructors in early 2023 about their perceptions of ChatGPT and LLM technologies. We study beginning students and report on student perceptions from our post-experiment survey and interview.

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. [64] present new interfaces for Visual Studio that present code changes. Liu et al. [43] build a new interaction model, grounded abstraction matching [43], 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 code-to-text. The concept of using natural language for coding has a long history [47] and has led to numerous ideas about bringing programming closer to how users communicate [50]. For instance, Hindle et al. [25] imagined that future language models could be effective at turning natural language to code, a prediction that has been borne out with Code LLMs.

Recent work has explored how Code LLMs could impact non-experts programmers: Finnie-Ansley et al. [19] reports 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. [14] compare the quality of Codex-generated code to student-written code; Babe et al. [3] use student-written prompts to benchmark Code LLMs; Leinonen et al. [36] report that Code LLMs are better at explaining code than beginning students; and Leinonen et al. [37] and Phung et al. [54] show that LLMs can help generate better error messages than traditional approaches. 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, 11, 17, 20, 29, 38, 49, 51, 54, 59, 62, 68]. The aforementioned papers present new tools, benchmarks, and studies of LLM capabilities. But, they do not study users, 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 in this area are particularly relevant to our task.

Zamfirescu-Pereira et al. [70] study non-experts prompting an LLM to produce recipes. Their participants actively avoided systemic testing, which we address in our experimental design by automating testing. Like them we find that participants' mental models of LLMs are very different from how they actually work.

Singh et al. [63] compare user interactions with a multimedia writing interface and a blank page editor. An LLM architecture delivers audio, text, and image suggestions to the user while writing. Our work was inspired by their focus on participant's perceptions of AI and system interaction.

3 STUDYING CODE LLMS AT MULTI-INSTITUTIONAL SCALE

We present a large-scale, multi-institution [18] study of how beginning programmers write natural language prompts for Code LLMs. We design a controlled, web-based experiment that allows participants to focus entirely on the *natural language to code task*. This allows us to dive deeply into the strategies that participants develop to author and edit natural language prompts.

Why CS1 students? When studying interactions with Code LLMs, it is important to select tasks at the right difficulty level for participants. We select a population of participants with a standardized level of Python experience: college students who have completed the first introductory CS course but not the second. We use problems that are similar (or identical) to the problems they did in CS1; consequently, participants should understand our tasks and even know how to solve them in code.

We recruit participants from three U.S. institutions: an R1 university (R1), a small liberal arts college (SLAC), and a women's college (Women's College). This increases the likelihood that our findings will generalize across institutions. The number of participants (n = 120) exceeds previous in-the-wild aor nd lab-based studies, providing an at-scale perspective. Our participants have similar training in programming, but are diverse in other ways: previous mathematical coursework, extracurricular programming, language exposure, first-generation college student status, and more. This allows us to investigate how different groups of students perform on the text-to-code task.

Why a lab-based experiment? Working with a Code LLM in a naturalistic setting involves 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 study limits user interactions to just writing natural language prompts* in order to isolate prompt writing and editing strategies. Unlike previous work [6, 30, 55, 65], we do not allow users to prompt with code or edit generated code. This gives us a large sample (n=2569) of natural language prompts, allowing us to explore in depth how participants write and edit prompts. It also ensures that all participants are engaging with the model: in open-ended tasks, a participant can edit code instead of modifying their prompt or even ignore the model and write code directly.

In order to further isolate the prompt writing and editing steps, we automatically test the LLM-generated code for participants. This reduces the likelihood that they will accept incorrect code instead of modifying their prompts. We also control how they interact with the model, isolating the interaction away from IDEs, educational or otherwise, which provide additional support or distractions.

How did we select problems? We selected programming problems from CS1 courses at the three institutions. We made some small changes to facilitate testing (e.g., returning output rather than printing). We balanced problems across eight conceptual categories (Figure 2), allowing us to explore which programming concepts are more challenging in a text-to-code setting. We selected six problems from each category, for a total of 48 problems. To better balance difficulty and category coverage, previous CS1 instructors were asked to provide additional problems. Each individual problem was assigned to 20 students, allowing us to explore prompting strategies at a per-problem level. We balance the experimental lists to control for ordering effects.

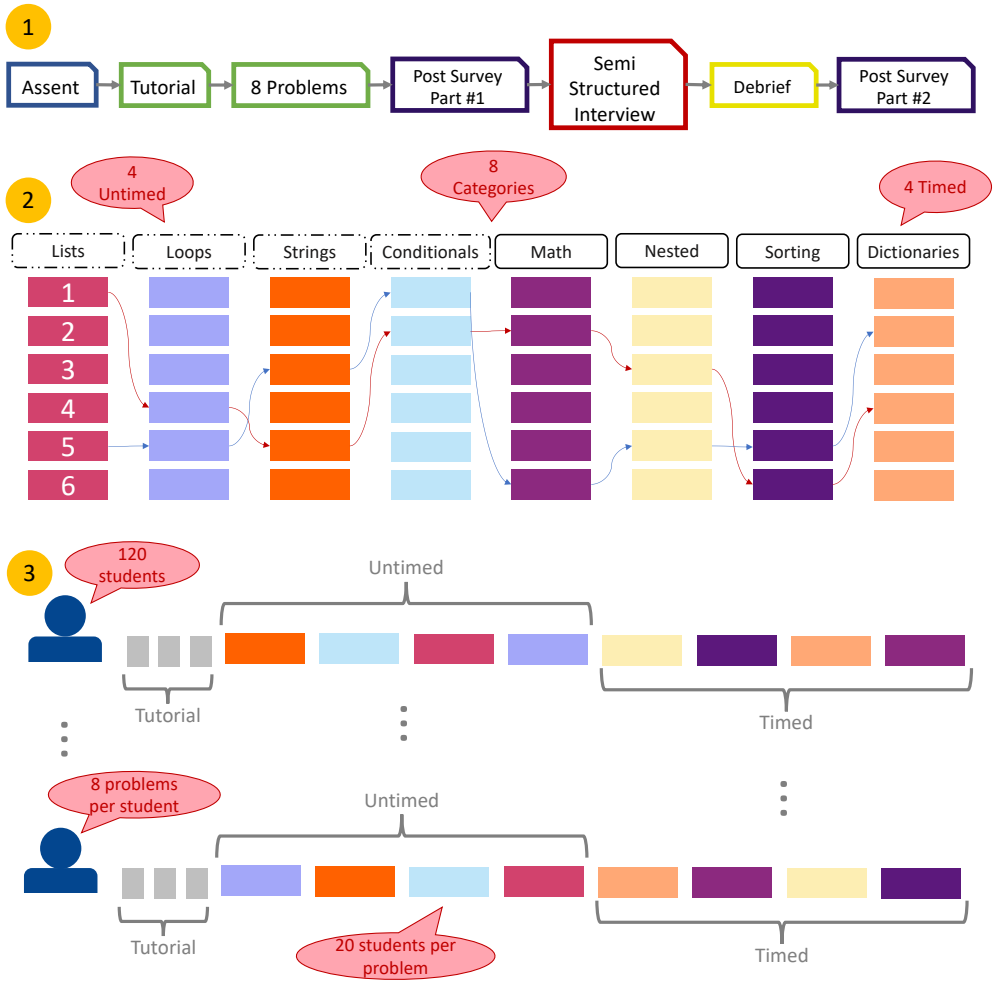
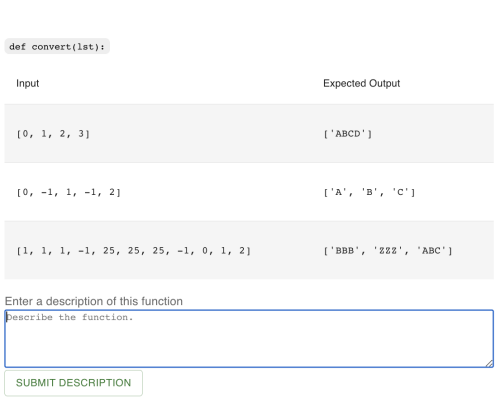
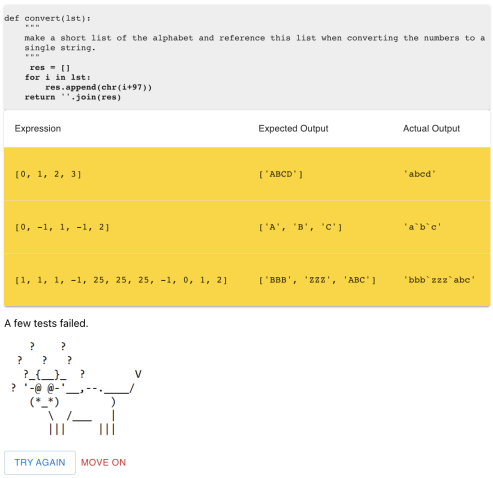


Fig. 2. Study overview. (1) describes the overall student trajectory through the study. We split the post survey into two sections, divided by the semi-structured interview, to delay collecting demographic information to prevent self-bias. (2) outlines the 8 problem categories (4 timed versus 4 untimed) 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.

How did we present problems without giving away a solution? A significant challenge in previous studies has been describing tasks to participants without biasing how they interact with the LLM or giving away the solution. For example, Kazemitabaar et al. [30] found that 32% of prompts that students entered into their LLM-powered system were exactly the natural language descriptions they were given. We address this issue by describing tasks in terms of input/output pairs. Our experiment shows the function and argument names along with three to five input/output examples (Figure 3). Participants have to write descriptions entirely in their own words, but can refer to function and parameter names in their descriptions, as is common in documentation.



(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.



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

Fig. 3. The *Charlie the Coding Cow* interface.

How did we validate problems? We validated each problem to ensure it was solvable, but not trivially so. First, we solved problems ourselves using the Code LLM to ensure that a working natural language description exists. Second, we checked that the model could not solve problems from their names alone (as is possible for common functions like factorial). Finally, to address the nondeterminism of Code LLMs, we ran each validation check multiple times to obtain a stable estimate of pass rates (§6.2).

How did we select and configure the LLM? When we began piloting in November 2022, the most capable Code LLM was the largest Codex model from OpenAI, `code-davinci-002`. Several capable LLMs have since appeared, including non-proprietary LLMs that are better for reproducibility (§9). Instruction-tuned models that support conversational AI, such as ChatGPT, also appeared after we started. For consistency, we used the same Codex model throughout the study. 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 [12].

4 STUDY DESIGN

In §3 we describe our multi-institutional experimental design motivation. In this section, we discuss the logistics of participant recruitment and study execution.

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. If the code fails, they can retry the problem or move ahead. For retry attempts, we pre-fill the text box with their last prompt to make editing easier. Finally, after every final attempt 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?*.

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.

4.2 Participants

We recruited 40 participants from each institution (n=120). Eligible participants were at least 18 years old, had taken CS1 at their institution sometime between Fall 2021 and Spring 2023, and had not completed any subsequent CS course. We recruited participants from March-July 2023 until reaching our target 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.

4.3 Study Design Specifics

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.

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

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 set of questions in the post-survey were adapted from previous work [7, 16, 22, 23, 33, 66] asking about usability, trust in automation, AI perception, growth mindset, and cognitive load (see Supplemental

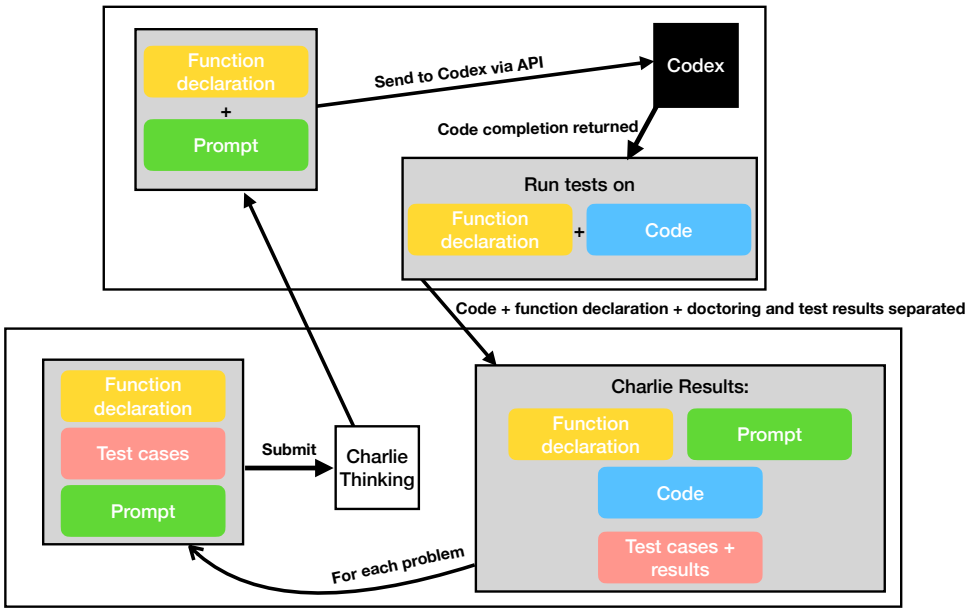


Fig. 4. A diagram of the Charlie system including the front-end interface, the back-end addition of prompts to the function declaration, sending declaration + prompt to Codex via the API, running the code completion from Codex on our pre-defined tests, and returning the results to the participant in the front-end interface.

Materials for more details). 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. 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 (see Supplemental Materials).

In late 2022, we ran a pilot study with 19 participants recruited from the three institutions to assess the study design and usability of the interface. The pilot led us to increase our time estimate and compensation for the main study.

5 ANALYSIS

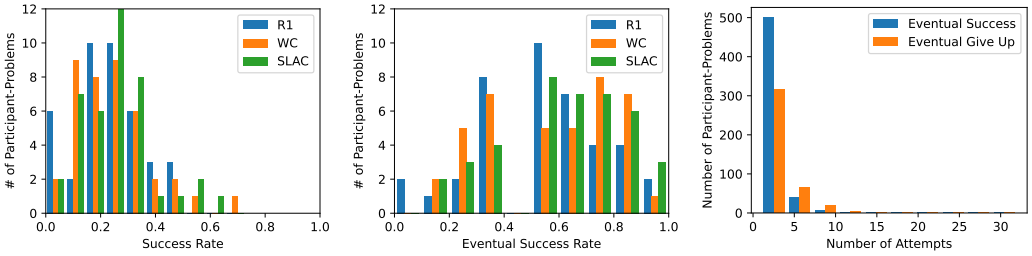
This section presents the analysis framework for §6, §7, and §8. We take a mixed-methods approach to this work.

5.1 Measures

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; details of the coding methodology and codebook can be found in the Supplemental Materials. We present selected quotes from the surveys and interviews throughout. Quotations have been lightly edited to remove speech errors and any identifiable information.Each participant's quote is accompanied by a pseudonym assigned to them during data collection.

Institution	Mean pass@1	Success Rate	Eventual Success Rate
SLAC	0.23	25%	61%
Women's College	0.23	25%	57%
R1	0.21	23%	54%
Overall	0.22	24%	57%

(a) Mean values of different measures of success.



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

Fig. 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.

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 Positionality

We are affiliated with the three institutions where this research was conducted; 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. Some authors are course instructors for CS1. As described in §4.2, 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.

Question	Scale	Mean
How mentally demanding was the task?	Very low->Very high	4
How hurried 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 1. Mean NASA-TLX ratings

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

6.1 Measures of Success

There are several ways to measure success at our task (writing natural language prompts for a Code LLM). One measure is the fraction of all attempts on which the model generates a working program; we refer to this as the *success rate*. A participant who succeeds after multiple failing attempts will have a lower success rate than one 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.

Although success rates measure the correctness of the code that students saw during the experiment, LLM generation is non-deterministic. To make our analysis more robust, we adopt a methodology that is commonly used to benchmark Code LLMs [12]. We compute *pass@1*: an estimate of the probability that an LLM will generate a working solution for a given prompt. The best practice for estimating pass@1 is to query the LLM 200 times for each prompt, test all generated programs, and use the estimation method of Chen et al. [12]. However, sampling 200 generations for all 2,569 prompts would be very expensive with the Codex model. Instead, we use a recently released open Code LLM called StarCoder [40] that is nearly as capable as the Codex model on Python benchmarks. Pass@1 rates 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.

6.2 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 participants struggled to write natural language prompts for the LLM. Moreover, many achieved success only after making repeated attempts at writing prompts. The challenging nature of this task is supported by comments from the students themselves (§7.1).

6.3 Do Participants Find the Task Challenging?

In the post-survey, participants completed four items of the NASA TLX [23]. Overall, students found the task mentally demanding (Table 1). 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

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	0.22
Raised Multilingual Including English	41	0.24
Raised Multilingual Not in English	2	0.20

Table 2. Self-reported high school, language, and family background.

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).

6.4 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 2 for a summary of demographics; details in Supplemental Materials). We find only two statistically reliable differences:

- **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 several other factors, but found no significant difference in pass@1:

- **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).
- **Computing intensive majors:** 42% of participants were pursuing computationally intensive majors. We observe identical pass rates for both computing and non-computing majors (0.22).
- **International students:** International and U.S. domestic students had the same 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 does not 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.

Thematic Codes	N
Charlie Doesn't Understand Me	89
Issues With Generated Code	59
Student Struggles	40
No Problems Mentioned	10
Issues with Study Platform	8
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?*

7 RQ2: WHERE DO STUDENT DIFFICULTIES COME FROM?

We have seen that students find it hard to prompt a Code LLM in natural language (§6) and now investigate why.

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=89); 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 was 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 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.

Category	Mean pass@1	Mean Eventual Success Rate	Student Difficulty Ranking
Sorting*	0.09	33%	1 (Hardest)
Dictionaries*	0.16	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.

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 student interview responses to questions 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 pass@1, with fixed effects of category, institution, and their interaction, and random effects of problem and participant. A statistically reliable difference in pass@1 was observed for Sorting problems, which were the most challenging (p=0.045). Participants from SLAC had lower pass@1 in the Nested category compared to other students, but the effect was not reliable (p=0.063).

Student interviews provide insight into their post-task perspectives. Our analysis yielded response classification by the eight categories. 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 [27] or Python 2 print statements. These are model failures.

Declaration, Prompt & Completion	`def exp(lst, val):`
	`'''`
	`Multiply each number in the list by the exponent of the given value.`
	`'''`
	`return [i ** val for i in lst]`

Question	Is this code you would write yourself?
Student Responses	Women's College: Yes, SLAC: No

Fig. 6. An example problem, exp, which was rated differently by SLAC and Women's College students, likely due to the list comprehension.

- 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 (§3), 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*

Thematic Codes	N
Process: Sequential	13
Process: Translation	12
Knowledge: Keywords - General	29
Knowledge: Keywords - Database/Dictionary	16
Knowledge: ChatGPT	17
Knowledge: Copilot/Codex	2
Knowledge: Internet Data	12
Knowledge: Intermediate Representation	4
No Guess	13
N/A	23

Table 5. Thematic codes emerging from responses to *How do you think Charlie works?*

that operator." Others mentioned `map`, `replace`, and `try`/`except`. List comprehensions are an interesting case because Women's College teaches them, but SLAC does not. When asked about generated code with list comprehensions, 9/24 (37.5%) SLAC students indicated that it is similar to code they would write themselves, compared to 20/33 (60.6%) Women's College 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 investigates 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=12) 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
Added Detail	48
Added Coding Language	21
Reordered Prompt	5
Removed Detail	4
Fixed Grammar	2
Looked at Tests First	30
Looked at Code First	29
Looked at Code and Tests Together	8
Reread the Problem	7
Ran Prompt Again	3
Comment Not Relevant	16

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.*

Charlie's Knowledge. Most students hypothesized that Charlie relies on keywords (n=45). A large group of students (n=29) had a vague keyword mental model. For instance, *"I guess he probably looks for key words, "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 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, 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*

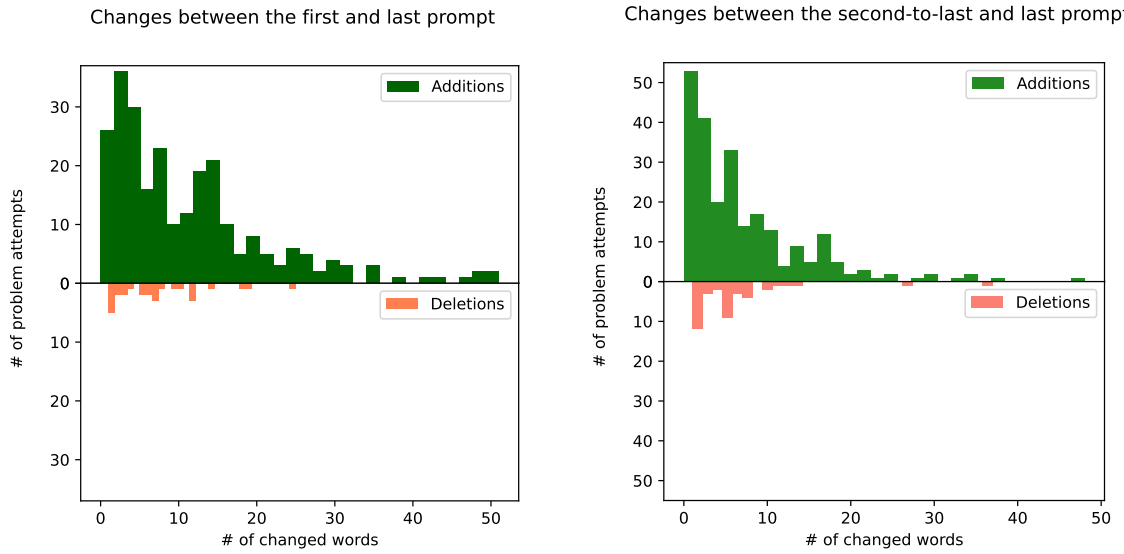


Fig. 7. Histograms of the 282 prompts which lead to successes after 2 or more attempts. These represent trends in how students edit prompts. The figure on the (left) shows the number of words changed between a first prompt and last prompt. The figure on the (right) shows the final change that produces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

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: *"At first [...] I 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"*, 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."*

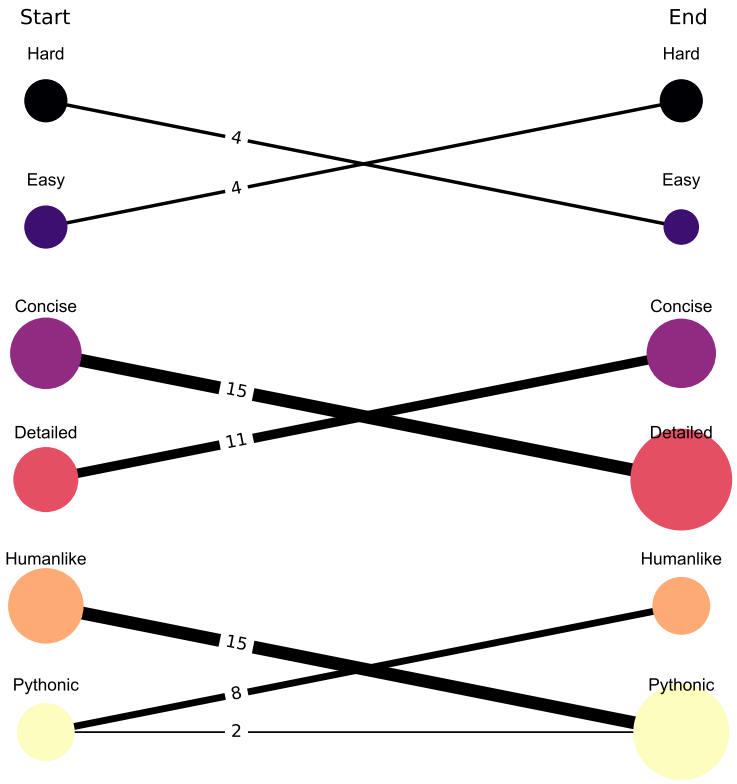


Fig. 8. Visualization of how students describe their editing trajectories. The left nodes represent how students described how they began their process. The right nodes represent how students described how they edited prompts at the end of the study. The codes are presented in pairs - Hard versus Easy, 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.

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.

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 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) and correlation with success rate. * indicates statistical significance.

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 [49, 71], even when no concrete productivity or correctness benefits are observed [65, 69]. 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. [66] and Druga and Ko [16] 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 [33]'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 Supplemental Materials 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*

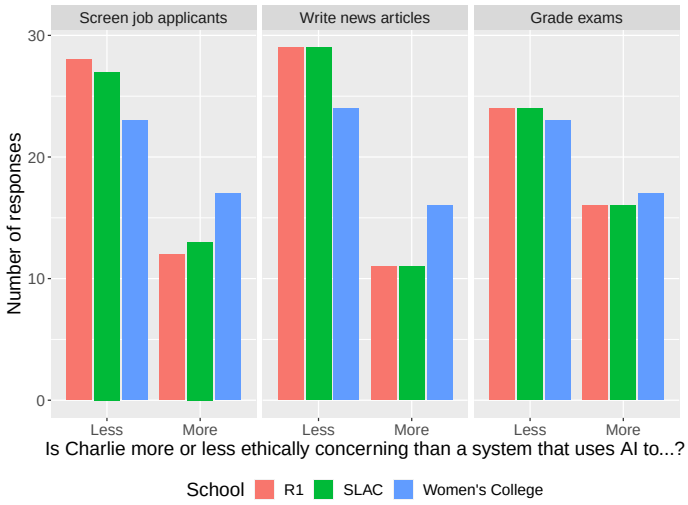


Fig. 9. Student perceptions of Charlie's ethicality as compared to other AI scenarios

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 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 this section we draw connections between our findings 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" [46, 67]. 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 provided tests to validate LLM output

automatically. Despite the fact that all our participants passed CS1, there is no evidence that this directly translates into their ability to write natural language prompts for a Code LLM.

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 the model in the long term. Existing work on experts manages to isolate different "modes" of interaction for different types of work [6]. Our findings suggest that near-novices *do not develop well-defined strategies for how to prompt*. Students added more detail to their previous prompts, even when they would do better starting from scratch. In addition, students' prompting abilities did not observably improve during the study (§8.3). These results suggest that prompting, like most ways of interacting with code, needs to be explicitly taught to be used effectively.

Our findings also have implications for complete novices. Figure 8 outlines how students described their start and end approaches to editing. We see that many students who started out writing prompts as for a human transition into using more coding terminology by the end of the study. No students who reported starting with humanlike descriptions ended that way. Participants therefore identified a key property of Code LLMs: they are trained primarily on expert-written code and documentation and thus expect natural language prompts to utilize coding terminology. This is not a skill that complete novices possess.

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 were aware that they were interacting with an AI code generation tool, and the majority (n=88, 73% in the post survey) indicated that they 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, similar to dictionaries that they learned in their CS1 course.

These mental models cannot explain one of the aspects of Codex that students find the most 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 retrieval mental model of Code LLMs that most students in our study adopted. Without a well-developed understanding of why this happens, students have simply added another unknown computational behavior to their coding experience.

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* [14, 19]. Our findings show that although Code LLMs can solve CS1 problems, CS1 students cannot necessarily use 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. 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 SLAC 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 questions [13, 45], which many educators see as valuable, but difficult to use without automation [21].

On the other hand, a sizeable number of students did not support using Code LLMs in CS1. Students brought up concerns related to ethics and to potential harms of diminished knowledge or sense of fulfillment if they came to rely on AI. Our findings raise concerns about equity: we find that students with extracurricular programming experience have an advantage on the task. We also find that first generation college student-written prompts have reliably lower pass@1 rates than those of other students. Educators should weigh the potential benefits of adopting this new technology against the possibility that it might exacerbate existing equity issues [26].

Finally, we find our students are ambivalent towards AI systems. 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 as the backend for Charlie. While Codex provided state-of-the-art Code LLM performance, it came with critical downsides and limitations. In the middle of our experiment, 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 [40, 60] 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 [39], 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 will become less usable and less available to the majority of people, adding to the wide ranging list of ethical concerns about generative AI [9, 31, 42].

10 THREATS TO VALIDITY

A major challenge of studying AI-human 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 §6.4. 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. Finally, novelty bias is always a potential concern when evaluating novel interfaces or systems.

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.

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.

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 arXiv:2301.11325 [cs, eess].

[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. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 6470–6484. https://doi.org/10.18653/v1/2020.emnlp-main.525

[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 arXiv:2306.04556 [cs].

[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. 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 (2009), 71–81. Publisher: Springer.

[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

[11] Le Chen, Xianzhong Ding, Murali Emani, Tristan Vanderbruggen, Pei-hung Lin, and Chuanhua Liao. 2023. Data Race Detection Using Large Language Models. arXiv:2308.07505 [cs]

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

[13] 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

[14] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming Jiang. 2022. GitHub Copilot AI pair programmer: Asset or Liability? *ArXiv* abs/2206.15331 (2022).

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

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

[17] Kasra Ferdowsi, Ruanqianqian Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. 2023. Live Exploration of AI-Generated Programs. arXiv:2306.09541 [cs.HC]

[18] 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

[19] 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 event-place: Virtual Event, Australia.

[20] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (6–8). San Fransisco, CA, USA.

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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.

[26] Kenneth Holstein and Shayan Doroudi. 2021. Equity and Artificial Intelligence in Education: Will" AIEd" Amplify or Alleviate Inequities in Education? *arXiv preprint arXiv:2104.12920* (2021).

[27] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rygGQyrFvH

[28] Daphne Ippolito, Ann Yuan, Andy Coenen, and Sehmon Burnam. 2022. Creative Writing with an AI-Powered Writing Assistant: Perspectives from Professional Writers. arXiv:2211.05030 [cs.HC]

[29] 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

[30] 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

[31] 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].

[32] 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

[33] 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.

[34] 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 V.1*. ACM, Chicago IL USA, 106–121. https://doi.org/10.1145/3568813.3600138

[35] 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 Number: 4.

[36] 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

[37] 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

[38] 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

[39] 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

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

[41] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2023. Understanding the Usability of AI Programming Assistants. arXiv:2303.17125 [cs.SE]

[42] 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].

[43] 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

[44] 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 arXiv:2304.08551 [cs].

[45] 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

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

[47] 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

[48] 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

[49] 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].

[50] 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

[51] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2023. In-IDE Generation-based Information Support with a Large Language Model. arXiv:2307.08177 [cs.SE]

[52] 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

[53] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590 [cs.SE]

[54] Tung Phung, José Pablo Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Kumar Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. *ArXiv* abs/2302.04662 (2023).

[55] 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

[56] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, and others. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[57] 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.

[58] 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

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

[60] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, 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. 2023. Code Llama: Open Foundation Models for Code. http://arxiv.org/abs/2308.12950 arXiv:2308.12950 [cs].

[61] 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

[62] 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 arXiv:2302.06527 [cs]

[63] 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

[64] 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)*.

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

[66] 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*. 1–14.

[67] Matt Welsh. 2022. The End of Programming. *Commun. ACM* 66, 1 (dec 2022), 34–35. https://doi.org/10.1145/3570220

[68] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal Fuzzing via Large Language Models. arXiv:2308.04748 [cs]

[69] 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*. 1–10.

[70] 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

[71] 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.