

# How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment

Majeed Kazemitabaar  
Department of Computer Science,  
University of Toronto  
Toronto, Ontario, Canada  
majeed@dgp.toronto.edu

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

Austin Henley  
Microsoft Research  
Redmond, Washington, USA  
austinhenley@microsoft.com

Barbara J. Ericson  
School of Information, University of  
Michigan  
Ann Arbor, Michigan, USA  
barbarer@umich.edu

David Weintrop  
College of Education, University of  
Maryland  
College Park, Maryland, USA  
weintrop@umd.edu

Tovi Grossman  
Department of Computer Science,  
University of Toronto  
Toronto, Ontario, Canada  
tovi@dgp.toronto.edu

## ABSTRACT

As Large Language Models (LLMs) gain in popularity, it is important to understand how novice programmers use them and the effect they have on learning to code. We present the results of a thematic analysis on a data set from 33 learners, aged 10-17, as they independently learned Python by working on 45 code-authoring tasks with access to an AI Code Generator based on OpenAI Codex. We explore several important questions related to how learners used LLM-based AI code generators, and provide an analysis of the properties of the written prompts and the resulting AI generated code. Specifically, we explore (A) the context in which learners use Codex, (B) what learners are asking from Codex in terms of syntax and logic, (C) properties of prompt written by learners in terms of relation to task description, language, clarity, and prompt crafting patterns, (D) properties of the AI-generated code in terms of correctness, complexity, accuracy, and (E) how learners utilize AI-generated code in terms of placement, verification, and manual modifications. Furthermore, our analysis reveals four distinct coding approaches when writing code with an AI code generator: *AI Single Prompt*, where learners prompted Codex once to generate the entire solution to a task; *AI Step-by-Step*, where learners divided the problem into parts and used Codex to generate each part; *Hybrid*, where learners wrote some of the code themselves and used Codex to generate others; and *Manual* coding, where learners wrote the code themselves. The *AI Single Prompt* approach resulted in the highest correctness scores on code-authoring tasks, but the lowest correctness scores on subsequent code-modification tasks during training. Our results provide initial insight into how novice learners use AI code generators and the challenges and opportunities associated with integrating them into self-paced learning environments.

We conclude with various signs of over-reliance and self-regulation, as well as opportunities for curriculum and tool development.

## CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**; • **Social and professional topics** → **Computing education**.

## KEYWORDS

Large Language Models, OpenAI Codex, ChatGPT, Copilot, qualitative analysis, introductory programming, self-paced learning

## ACM Reference Format:

Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. In *Proceedings of Koli Calling International Conference on Computing Education Research (Koli Calling)*. ACM, New York, NY, USA, 12 pages. <https://doi.org/XXXXXXX.XXXXXXX>

## 1 INTRODUCTION

Large Language Models (LLMs) trained on code like OpenAI Codex [12] are capable of generating functioning programs from natural language descriptions. Since publicly made available by companies like OpenAI through user-facing tools such as ChatGPT (a Q&A chatbot) or Github Copilot (an IDE-based AI coding assistant), the code generation capabilities of LLMs are becoming more accessible to a wider array of people. These tools have the potential of scaling up computing education in self-paced learning environments and broadening participation in computing by assisting beginners with debugging, code generation, code explanation, and responding to questions about code.

Despite their potential benefits, LLMs present challenges in educational contexts. Their usage could result in learner dependency, hindering code authorship without assistance. Novice coders may also struggle with technical jargon, expressing coding intent, and comprehending or verifying AI-generated code. Additionally, from an educator's perspective, issues around academic integrity and plagiarism pose valid concerns [5].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*Koli Calling, 2023, Finland*

© 2023 Association for Computing Machinery.  
ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00  
<https://doi.org/XXXXXXX.XXXXXXX>

While LLM-based code generators are not specifically designed for education contexts, recent studies have assessed their performance in computer science education and their effects on learners and teachers [15, 17, 18]. Kazemitabaar et al. [28] compared students' learning outcomes with and without an AI code generator based on OpenAI Codex. They found that using Codex did not harm learners' performance on post-tests one week later, and students who performed better on Scratch pre-tests performed significantly better on retention post-tests if they had prior access to Codex. However, we still do not know how these outcomes occur. To fully understand the benefits of AI code generators in education, it's crucial to know how novices use them while learning to code [5, 18].

In this paper, we conduct a thematic analysis on a data set of 33 novice learners (ages 10-17) who had access to an AI Code Generator while learning Python programming for the first time. We attempt to answer two main research questions:

- **RQ1:** How do novices use and interact with LLM-based Code Generators when learning to write code by practicing CS1 coding tasks in a self-paced learning environment? **A:** When do learners use Codex in the problem-solving process? **B:** What are learners asking Codex to generate? **C:** What are the properties of the prompts that novices craft in terms of relationship to the task description, language, and vagueness? **D:** What properties does the AI-generated code have? **E:** How do novices use, modify, and verify AI-generated code?
- **RQ2:** What coding approaches do novice learners employ when they have access to LLM-based code generators to solve programming problems? How do these approaches differently impact learning outcomes measured by retention post-tests?

To answer these questions, we analyzed log data from a previous study [28] using a custom log analysis tool. Our findings reveal when and how novice learners interact with LLM-based code generators in addition to four distinct coding approaches that reflect learners' personal choices for utilizing AI code generators during programming. Our thematic analysis reveal various signs of over-reliance and self-regulation in students when they write code with an AI code generator. Our results suggest the importance of effective usage patterns for maximizing learning outcomes when novice learners have access to AI code generators. These findings can inform the design of future introductory programming tools, and highly scalable self-paced learning environments that incorporate AI code generators and the pedagogy that accompanies them.

## 2 RELATED WORK

### 2.1 Programmers' Experience with AI Coding Assistants

Large language models (LLMs) are deep neural networks trained on extremely large-scale model parameters using terabytes of textual data sets. When trained on large corpora of source code, these models can also generate code from natural language descriptions [2, 12, 23]. Pre-trained LLMs, like OpenAI Codex [12] based on GPT-3, Google PaLM [13], and DeepMind AlphaCode [32], have enabled LLM code generation tools like Github Copilot [20]. The

emergence of these technologies has sparked research on evaluating how professional programmers utilize AI code generators [45].

One line of research is exploring how users use GitHub Copilot [4, 36, 50, 51], a commercialized programming tool powered by OpenAI Codex. GitHub Copilot can provide autocomplete-style code suggestions based on existing code and natural language descriptions (e.g., comments). For instance, Barke et al. [4] conducted a grounded theory analysis with 20 programmers to identify two ways in which programmers interact with Copilot: acceleration, where Copilot speeds up the writing of code in "small logical units" because the programmer has a clear intention, and exploration, where Copilot suggestions are used to help with planning and exploration. There also have been other studies analyzing programmer experiences with more experimental AI coding assistants [25, 53, 54]. For example, Ross et al. [44] developed a conversational-based coding assistant and found two main usage patterns: bringing assistant's help to solve the entire challenge at once, or breaking it down to solve each smaller task individually.

However, most of the existing usability work on AI-assisted coding assistants focuses on the experience of expert programmers within well-designed experimental context, typically by testing a small number of programming tasks. To our knowledge, only Jayagopal et al. [24] focused novice programmers' experience with different program synthesizers including Copilot, but only for three tasks. To further elucidate this topic, we believe it would be enlightening to understand novice programmers' interaction with such AI code generators in an authentic programming context with a larger number of coding tasks. Therefore, in this work, we would like to analyze the user interactions collected in a data set of novice learners as they make progress and learn Python programming in a self-paced learning environment while they have access to an AI code generator.

### 2.2 Large Language Models in Computer Science Education

As LLMs become more widely used in practice, education researchers are exploring the potential of LLMs to produce educational content, enhance student engagement and customize learning experiences [27]. Recent work in the computer science education community has started to explore the implications and opportunities of LLMs on computer science learning from different perspectives [5]. Most of the recent work focused on understanding the capabilities of LLMs for completing programming tasks [15], generating instructional content [31], and developing new content creation methods [16]. For example, Finnie-Ansley et al. [18] showed that Open AI Codex performs better than most students on code writing questions in both CS1 and CS2 exams. When it comes to creating CS educational content, prior evaluations showed that LLMs have the potential be used to produce high enhancements on programming error messages [31] and provided high-precision feedback on code for fixing syntax errors [39]. Furthermore, Sarsa et al. [46] analyzed the novelty, plausibility, and readiness of 120 programming exercises generated by OpenAI Codex and proposed the potential of using such models to come up with coding assignments. As for code explanations, MacNeil et al. [32] reported student experience with LLM-generated code explanations in a web software

development e-book. They showed that most students perceived the code explanations as helpful, but engagement depends on code length, code complexity and explanation types.

To better integrate LLMs into computer science education and achieve student-centered learning, it is important to understand how novice students use and interact with AI code generators while they learn to code. For instance, we need to know what usage patterns and coding approaches students employ when learning with AI code generators, how these patterns and approaches impact their learning, what types of prompts students write, what are the attributes of the generated code and its relationship to the prompts, and how students integrate AI-generated code into their existing code. In this paper, we investigate these questions through a systematic analysis of student log data on their interaction with OpenAI Codex.

### 2.3 Supporting Novices While Writing Code

To help students write code, previous research has explored a variety of scaffolded approaches. Bruner [9] coined the term "scaffolding" to refer to the process of giving students support structures so they can learn a subject or skill that is above their level. Appropriate scaffolding will equip learners with sufficient knowledge and skills to perform the task independently. One direction provides support before writing the actual code. For example, flowcharts have been used to brainstorm and organize solution ideas before diving into coding, according to Renske & Sjaak [48]. This has resulted in an improvement of algorithm design and programming skills. Moreover, Cunningham et al., [14] described a multi-stage programming process including arranging plans explicitly.

Another direction is to provide immediate assistance during the actual code writing process. Such immediate assistance can be done by providing detailed feedback based on the student's current code states [29]. This feedback could be explanations on what is wrong in the existing code [47], suggestions on how to fix the error [49], or next-step hints on improving the student's solution towards the goal [43]. In addition, when students ask for help, the available support could also be a library of worked examples that students can run and modify with output [52] or an equivalent Parsons problem that students can actively work on [21]. While previous research has examined a wide range of supportive methods, none of them have considered the implications of AI code generator to support students to write code.

As a first step, Kazemitabaar et al. conducted a controlled experimental to investigate the effects of incorporating an AI code generator during code writing compared to a baseline condition (writing code without any assistance) [28]. The previous work highlighted the need for a systematic analysis of students' interaction with AI code generators to better understand their impact on learning. Building upon this prior research [28], our aim is to identify key moments when students choose to use AI code generators, the attributes of the code they seek in different situations, the language properties used in prompts, how they utilize AI-generated code, and how the coding approaches they use with AI code generators, influence their learning.

## 3 METHODOLOGY

### 3.1 Data Set and Data Instrumentation

To explore how novice use AI code generators, we analyzed a data set from a prior study [28] in which 69 novice learners (ages 10-17) used Coding Steps (<https://github.com/MajeedKazemi/coding-steps>) as part of a three-week study. Coding Steps is a self-paced, online learning environment that includes 45 CS1 Python coding tasks that were designed to gradually introduce new concepts. The system includes an embedded programming environment, functionality to submit code to remote instructors that grade submitted work and provide real-time personalized feedback, a novice-friendly Python tutorial, and an AI code generator (based on OpenAI Codex).

The original study included two conditions: one in which learners had access to an AI code generator (the Codex condition), and another in which the AI code generator was not available (the Baseline condition). In this work, we reuse the same data but only focus on the log data from the 33 learners in the Codex condition to investigate new research questions. Four types of timestamped logs were collected from learners: code edit logs, console run logs, AI code generation logs (including prompt message and generated code), and submission logs. Log data has become an important data source to understand programming experiences [8] and coding approaches [18, 22]. Log data can provide valuable insights for computing education researchers [34] as it can capture detailed information about students' experiences and actions.

### 3.2 Original Study Design and Participants

The original study consisted of ten 90-minute sessions, held over three consecutive weeks, covering a session of introduction to Scratch, seven sessions of Python training, and two evaluation sessions including a retention post-test one week later.

**3.2.1 Participants.** The original study included 69 participants (21 female, 48 male) aged 10-17 ( $M=12.5$ ;  $SD=1.8$ ). Participants were recruited from over 200 sign-ups at coding camps in two major North American cities. The study was approved by the Research Ethics Board of the original author's institution. Parental/guardian consent was obtained before the first session and each participant received a \$50 gift card as compensation.

From the 33 participants that we focus on in this study, none reported any prior text-based programming experience, 32 indicated using a block-based programming environment like Scratch or Code.org, and 12 indicated taking a programming-related class in the past. In terms of language, 25 participants were English speakers, while 1 reported difficulty explaining things in English. More details about the study can be found in the paper that presents the original study [28].

**3.2.2 Study Procedure.** The first session included a one-hour lecture on the fundamentals of programming using Scratch, and a pre-test evaluation including 25 multiple-choice questions about Scratch programming. The Scratch lecture covered all the key concepts required in the training phase, including: input/outputs, random numbers, arithmetic operators, conditionals, comparators, logical operators, loops, and arrays.

Students then began the training phase by watching an introduction video with several examples of AI code generation. As

a self-paced online learning environment, learners received personalized feedback from remote instructors and were provided a novice-friendly Python documentation that included worked examples, common Python errors, and debugging strategies. During this phase, learners worked on 45 two-part programming learning tasks and 40 multiple-choice questions at their own pace using the Coding Steps IDE (Figure 1). The tasks and the topics were presented in a fixed order with gradually increasing complexity. For more details about each of the tasks, read the Appendix A of the original paper [28].

Each coding task had two parts: code-authoring and code-modification. For each part, learners first read the task description and a few examples of input and expected output. To finish a code-authoring task, learners were instructed to write code in the code editor with access to the AI code generator. Learners were also provided with a Python documentation inside the IDE. Regarding the code-modification tasks, learners were given their own accepted submission from the code task as the starting point. If the learner skipped a code-authoring task, they were shown an example correct solution as the starting point for the associated code-modification task. Learners had no access to the AI code generator for modification tasks.

Finally, the evaluation phase comprised an immediate post-test followed by a retention post-test administered one week later. The tasks used in the evaluation phase were analogous in terms of difficulty and topics to the tasks used in the learning phase. Both post-tests contained 10 coding tasks (5 code-authoring and 5 code-modification tasks), and 40 multiple-choice questions. Learners had no access to the AI code generator, Python documentation, or instructor feedback during the evaluation phase.

### 3.3 Data Analysis

To assist with our data analysis, we developed a web interface for visualizing log data and replicating student behaviors in a vertical time sequence based on log entries. The interface displays high-level metadata for each task, keystroke counts for edit actions, side-by-side diffs for code modifications, prompt messages along with AI-generated code for Codex usages, and console input/output for execution actions. When analyzing data, we applied a combination of deductive and inductive approaches in thematic analysis [6, 35]. We applied a deductive approach to categorize learners' log data into groups based on task number and student ID, then we analyzed each student-task pair under two levels.

For analysis, we defined each AI code generation request (which was initiated by a prompt) as a single unit of analysis, resulting in a total of 1666 data points. To answer RQ1, we created four high-level code dimensions: (i) the context of using Codex in terms of editor contents and prior actions (RQ1 A), (ii) prompt attributes including requested content, details, and language (RQ1 B, C), (iii) quality of the AI-generated code (RQ1 D), and (iv) utilizing AI-generated code (RQ1 E). This enabled us to focus on relevant data for each research question in following rounds of analysis [7]. Additionally, to answer RQ2, we derived four high-level recurring coding approaches based on RQ1 and then assigned a single approach to each of the tasks submitted by learners.

Under each high-level dimension, we applied an inductive approach where we read through all the data and allowed codes to emerge during the process [7]. We generated the codebook iteratively following this process: First, two researchers (the first two authors of this paper) familiarized themselves by going through all the logs, identifying candidate sub-dimensions (if possible), and specifying codes for each dimension. These codes were then iteratively revised as the data was reviewed and drafted into an initial codebook. Moreover, the two researchers independently coded data for three tasks (6.4% of the data), each with different difficulty levels and topics using the initial codebook. After that, they discussed the initial coding results, resolved conflicts, and further refined the codebook.

After refining the codebook (Table 1), the two researchers independently coded another five tasks (13.2% of the data) and reached an inter-rater reliability of 0.87 (alpha values > 0.80 [37]) using percentage agreement [35]. After addressing the disagreements and finalizing the codebook, the two researchers coded the remaining data individually by assigning task numbers at random. The full codebook can be found at <https://tinyurl.com/codex-analysis-codebook>.

## 4 RESULTS

Overall, we analyzed 1666 Codex usages from 1379 submitted tasks (356 tasks were submitted without using Codex). A Codex usage is designated by writing a prompt in the AI code generator's text-box and pressing the generate button. The generated code is automatically placed at the user's cursor in the editor. We define the problem-solving process as a sequence of actions until the final submission of a task. Actions include manually editing code, using Codex, using the documentation, running the code, providing input to it, and submitting it for evaluation by remote instructors.

We denote participant numbers as  $P_n$  (ranked based on pre-test Scratch scores:  $P_1$  with the highest score).

### 4.1 RQ1 A: When do Learners Use Codex?

We identified five primary scenarios (based on the state of the editor) in which learners prompted Codex: at the beginning (46%,  $n=760$ ), after clearing the editor (5%,  $n=83$ ), after manual coding (17%,  $n=282$ ), after using Codex (34%,  $n=572$ ), and while already having the solution (1%,  $n=16$ ). Below we briefly describe each scenario.

**4.1.1 Starting with Codex.** In 760 tasks, learners used Codex at the beginning of the task, often (92%,  $n=703$ ) with no initial attempt of prior manual coding. Common behaviors included copying the full task description to generate the entire solution (66%,  $n=503$ ), rephrasing task description to generate the entire solution (7%,  $n=53$ ), and breaking down the task into subgoals (26%,  $n=199$ ).

**4.1.2 Using Codex After Clearing the Editor.** In 83 Codex usages, learners cleared their editor after being stuck with invalid code with repeated compiler errors and wrong edits (86%,  $n=71$ ), or failing to properly test correct code (14%,  $n=12$ ). Learners then cleared their editor and prompted Codex for either the entire (51%,  $n=42$ ) or part of the task (49%,  $n=41$ ). Notably, 75 of these 83 instances (90%) occurred after unsuccessful Codex usages, resulting in broken

**Table 1: High-level codebook that was used to analyze each Codex usage.**

| High-Level Code Dimensions                    | Sub-dimensions   |
|---|--|
| Context of Codex Usage (RQ1 A)                | <i>prior manual edits, prior codex usage, existing issues</i>  |
| Prompt Attributes (RQ1 B, C)                  | <i>request content, relationship to task, vagueness, requesting syntax/logic, fixing/localizing issue, repeated prompt</i> |
| AI-Generated Code Properties (RQ1 D)          | <i>quality properties, quality reason, generated extra unspecified code, outside curriculum</i>                            |
| Using and Modifying AI-Generated Code (RQ1 E) | <i>placement, modifying existing code, modifying AI-generated code, testing/verification, next action</i>                  |

AI-generated code or misplaced code that was challenging to fix manually.

**4.1.3 Using Codex After Manual Coding.** Out of 293 Codex usages, learners used Codex following instances of manual coding. In 102 (35%) instances, learners manually wrote correct code prior to prompting Codex. Learners typically wrote declarations (e.g., getting input from user or generating random numbers) and used Codex to implement the next major subgoal. Six instances involved learners writing complex code with loops or nested conditionals prior to prompting Codex. Furthermore, prior to 115 (39%) Codex usages, learners authored mostly correct code with minor issues. Common patterns included (i) struggling with string and integer concatenation before prompting Codex to fix, (ii) 28 instances of deliberately writing incomplete code like `num =` before prompting Codex with “random number”. In two unique cases,  $P_3$  manually wrote code that included ellipsis like `print("It took ", ..., "attempts.")` and then prompted Codex to fill in the behavior (e.g., “counting attempts”). In 76 (26%) instances, Codex was used after writing mostly incorrect code. Examples included syntax errors in code involving string and variable concatenation such as `join(name + "bot")` ( $P_2$ ), obtaining number from the user like `print "Enter a number:"` ( $P_{25}$ ). Learners used Codex to fix these issues.

Furthermore, in the 191 cases where learners faced issues after manual coding, Codex was used to fix existing code in 44% ( $n=85$ ) instances, generate the entire solution in 18% ( $n=34$ ) cases, or generate new code ignoring the issue in 32% ( $n=62$ ) cases. These initial manual coding attempts before using Codex highlight self-regulation in learners. Moving forward, it is crucial for future tools and curriculum to prioritize effective methods for using LLMs for debugging manually written code.

**4.1.4 Using Codex after Using Codex.** In 572 instances of using Codex after a previous Codex usage, learners handled the prior AI-generated code as follows: 53% ( $n=304$ ) kept it unchanged, 36% ( $n=206$ ) deleted it, 7% ( $n=41$ ) made minor modifications, and 4% ( $n=21$ ) broke it. Among the cases where learners correctly placed the AI-generated code (304 cases), 80% ( $n=243$ ) proceeded to generate new subgoals. Of interest, we identified 84 Codex usages in which learners requested code similar to what was already in their editor. Furthermore, after deleting the prior AI-generated code (206 cases), learners either re-attempted using Codex by rephrasing the prompt message in 40% ( $n=130$ ) cases, or decided to clear their entire editor (32%,  $n=66$ ).

**4.1.5 Using Codex while Having Solution.** We identified 16 instances where learners used Codex when they already had the task’s solution. Of interest, in six cases, learners compared their manually written solution with an AI-generated solution and performed minor edits to their own solution.

## 4.2 RQ1 B: What are Learners Asking from Codex?

Learners used Codex to generate the entire solution (43%,  $n=723$ ), generate new subgoals (37%,  $n=626$ ), and fix existing code (7%,  $n=110$ ). When generating new subgoals, we found 135 Codex usages where learners used Codex to generate code similar to existing code in their editor which may indicate over-reliance on the AI code generator. For example, when  $P_{25}$  prompted Codex with “ask for another number” they already had similar code in their editor: `number = int(input("Enter a number: "))`. Additionally, when fixing code using Codex, learners were often (61%,  $n=73$ ) able to correctly localize their issues. In seven cases, learners explicitly prompted Codex to fix their code (see Figure 1 for an example).

By analyzing crafted prompts, we identified two categories: requesting either **pure syntax** (e.g., “get a number from user”), or both **syntax and logic** (e.g., “get a number until it’s 0”). Of the 723 prompts where learners prompted Codex to generate the entire solution, 46% requested *syntax and logic*, while 54% sought *pure syntax*. However, in the 626 prompts for new subgoals, 85% asked for *pure syntax*, with the rest requesting *syntax and logic*. This demonstrates that learners mostly requested syntax when they decomposed a task into multiple subgoals.

## 4.3 RQ1 C: Prompt Properties

For each Codex usage, we analyzed the properties of the prompt in terms of its relationship to the task description, its language, and its clarity. Additionally, our analysis highlights two prompt crafting patterns: (i) solving a task by dividing its task description sentence by sentence for each prompt, and (ii) using repeated and slightly reworded prompts to achieve desired code.

**4.3.1 Prompt Relationship to Task Descriptions.** More than half (52%,  $n=864$ ) of all requested prompts were copied directly from the task description, with 27% ( $n=233$ ) being partial copies (requesting code for part of a task). The major groups of the remaining prompts (48%,  $n=802$ ) were written independently by learners: (i) 335 prompts were accurately reworded versions of the task description with all required details and using similar vocabulary, (ii) 223 prompts included less details or vague interpretations of the task



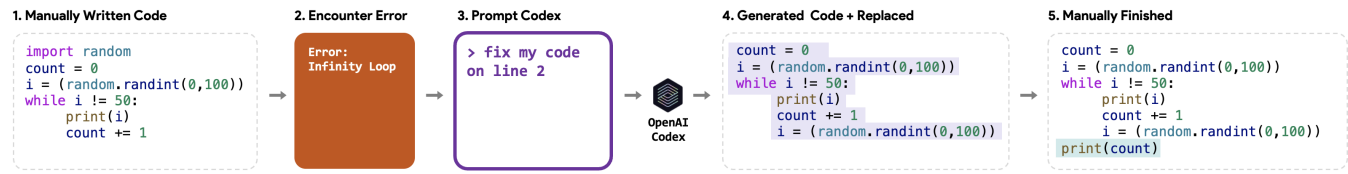


Figure 1: An example of using AI-generated code as an example to fix syntax error with writing loops.

which often led to code with omitted details or unintended behavior, (iii) 41 were based on incorrect task interpretations, and (iv) 71 unrelated to the task or Python programming.

**4.3.2 Prompt Language.** We identified 86 prompts written by learners that were similar to a pseudocode and included syntactical elements. For example, when  $P_4$  wanted to find the largest number in a list, they prompt Codex with a pseudocode that specified the exact logic “for  $n$  in numbers, if  $n > l$ , set  $l$  to  $n$ ” followed by “print Largest number:  $l$ ”. Compared to other learners who simply prompted the behavior (e.g., “find the largest number”), such usages of pseudocode indicate a deeper algorithmic thinking. Additionally, learners sometimes articulated their desired code by specifying the output format such as “display message Grade: grade”. We also observed prompts that described the entire behavior through an input/output example, including when  $P_{12}$  prompted “output: What is your name? output: Hello, Bob!” which generated the correct code. Furthermore, sometimes learners skipped static values in their prompt messages so they could focus more on the logic while crafting prompts. For example,  $P_6$  prompted “if variable number is greater than 75 print 2. If anything else print 3” and then modified the values of “2” and “3” in the AI-generated code based on the task requirements.

**4.3.3 Prompt Clarity.** We found that about 28% ( $n=201$ ) of prompts had some indicators of vagueness such as being under-specified. A common theme was not specifying the initial value for variable declarations like “variable called name” ( $P_8, P_{10}, P_{16}$ ). To generate random numbers, vague prompts did not indicate where to store the number, or the range to select from like “rand” ( $P_3$ ), or “number = roll” ( $P_{32}$ ). Vague prompts for conditionals did not specify the condition like “check variable” ( $P_{25}$ ). To define loops, vague prompts did not specify the stop condition like “forever, set  $M$  to input” ( $P_4$ ), or were expressed vaguely like when  $P_{20}$  prompted “go back to top” after noticing their code does not repeat.

**4.3.4 Prompt Crafting Patterns.** We found that learners frequently decomposed tasks into individual prompts. While effective for sequential and independent subgoals, this approach faltered when a sentence in the task altered or built on top of the instructions given in the previous sentence. In the latter case, the LLM usually regenerates the existing code with the added functionality, therefore, requiring learners to replace it with their existing code. Moreover, we examined tasks with multiple prompts and observed that 7% ( $n=109$ ) were exact repetitions, and 3% ( $n=55$ ) were slight rephrasings, typically used when initial prompts failed to yield the desired code. Only in 13 Codex usages learners added meaningful detail in reworded prompts. Notably, we found six instances where learners updated minor values via reworded prompts rather than modifying values in the pre-existing AI-generated code.

## 4.4 RQ1 D: AI-Generated Code Properties

We analyzed AI-generated code from three dimensions: correctness, relation to curriculum (complexity), and relation to prompt (accuracy). We found that of the 1666 Codex usages, 81% ( $n=1357$ ) of all AI-generated code was produced without any identifiable problems, while 19% ( $n=309$ ) of the generated code exhibited some problematic characteristics, including: not following task requirements 28% ( $n=87$ ), regenerating existing code 28% ( $n=86$ ), missing minor code 19% ( $n=60$ ), incorrect code 8% ( $n=25$ ), and generating only comments 8% ( $n=25$ ). Finally, we report on the relationship between the quality of AI-generated code and the quality of the prompts associated with them.

**4.4.1 Correctness of AI-Generated Code.** Here we focus on two problematic characteristics of AI-generated code. First, Codex regenerated code that existed in the editor in 86 usages. This behavior caused complications when learners had an erroneous code and used Codex to fix it. For example, when  $P_{15}$  wanted to fix their incorrect code `message = ("num is:" + num)`, they prompted Codex with “make a variable called message and define it so that it means num is: the num variable”, however, Codex regenerated their incorrect code. Furthermore, in 60 cases Codex did not generate minor pieces of code that were necessary for the code to execute properly, which was often (56 cases) a missing import random statement, or not casting input from string to integer.

**4.4.2 Complexity of AI-Generated Code.** We identified 22 instances in which Codex generated code that was either not covered in the curriculum or from advanced topics that were supposed to be introduced later in the training phase. For example, when learners have not reached tasks on loops,  $P_{26}$  prompted Codex with “user must enter a value” which resulted in Codex generating a while loop that repeatedly received an input from the user until it had a valid value. This is probably because the student included the word “must” in their prompt. Of interest, 11 usages in which the learner copied the task from the task description yielded into Codex generating code that was outside of the curriculum.

**4.4.3 Accuracy of AI-Generated Code.** We identified 204 cases in which Codex produced additional code that was unspecified in the prompt message, and instead was predicted from the editor’s content or the provided prompt message. For example, when the editor contained three variables called `num1`, `num2`, and operator,  $P_3$  prompted Codex with “check operator symbol” which generated four if-else conditions checking the operator variable with the basic math operators.

In 39 cases, the predictions of the extra code were first generated as comments followed by some code. For instance, when  $P_{25}$

prompted Codex with “*check if a number is divisible*”, it first generated `\# by 3` and then on the next line, `if number \% 3 == 0:`.

We analyzed the extra code that was generated in each of the 204 cases and compared it to the requirements of its corresponding task to determine the usefulness of the extra code. We found that 34% (n=70) were wrong and had to be deleted, 34% (n=60) were directly usable without no modification, 14% (n=29) were usable after some manual modifications, and 12% (n=25) were usable after minor changes to the values.

**4.4.4 Effect of Prompt Quality.** We analyzed the context, prompt, and the generated code to understand why Codex generated low-quality code. We identified reasons including poorly crafted prompts that do not properly articulate the intended behavior (n=105), prompts with missing important detail (n=71), and existing low-quality code in the editor causing Codex regenerate similar code (n=34).

Furthermore, by analyzing the source of prompts, out of 864 prompts that were copied from the task description, 81% (n=701) produced high-quality code, 7% (n=59) prompts produced extra unspecified code, 5% (n=41) produced code with a minor missing item, while 7% (n=63) were unusable as it either repeated existing low-quality code, generated only comments, or the code was not following the task requirements. From 354 prompts that were reworded versions of the task description, 72% (n=256) generated high-quality code, 10% (n=38) generated extra unspecified code, 2% (n=9) generated code with a minor missing item, while 15% (n=51) were incorrect, only comments, or were repeated incorrect codes. From the 225 prompts that were reworded versions of the task description but with less detail, 46% (n=104) generated high-quality code, 20% (n=66) generated extra unspecified code, 2% (n=4) with minor missing items, while 22% (n=50) prompts generated low-quality code that were not usable. From the 89 prompts that included pseudocode and syntactical elements, 63% (n=56) produced high-quality code, 16% (n=14) generated extra unspecified code, 20% (n=18) generated code that was not the intended behavior.

## 4.5 RQ1 E: Utilizing AI-Generated Code

Here we report how learners utilized AI-generated code in terms of *placement*, *verification*, and *modification*. We also present a pattern in which learners use AI-generated code as an example to fix their incorrect code.

**4.5.1 Placement of AI-Generated Code.** The AI code generator in Coding Steps generates code at the user’s cursor location. Therefore, learners had to manually adjust the code placement if their cursor was in the wrong spot. Additionally, when the AI-generated code included parts of the existing code, learners also needed to replace their existing code with the AI-generated code.

For the most part, learners were able to place the AI-generated code in the correct position. However, out of 1666 Codex usages, we identified 69 instances in which learners placed the AI-generated code incorrectly. There were seven cases of not placing the AI-generated code at the correct indentation (e.g., inside a loop), and six instances of placing the AI-generated code before a variable that it accessed was declared. Additionally, in 18 cases, instead of

replacing their original code with the newly AI-generated code, learners kept both versions.

**4.5.2 Verifying AI-Generated Code.** Through our thematic analysis, we identified three active verification approaches including tinkering with AI-generated code, properly running the code to evaluate it, and manually adding code for verification.

We identified 30 instances in which learners actively tinkered with the AI-generated code by temporarily changing values, conditions of while loops, modifying syntax, or even temporarily removing code to see how it contributed to the entire program. For example, after  $P_3$  declared a list of five elements, they prompted Codex with “*print 1st message in list*” and Codex generated `print(myList[0])`. They then tinkered with the list index, temporarily changing it from 0 to 1, testing the code to see how it affects the output.

From the total 1666 Codex usages, learners tested their AI-generated code in 60% (n=1005) usages. Of which, 71% (n=720) were executed correctly without any errors, and 29% (n=285) with errors or incorrect behaviors. However, after 485 Codex usages, learners did not run their AI-generated code to test it. In 65 cases (13%) learners deleted the AI-generated code, in 166 cases (34%) learners moved on to using Codex for the next part of the task, and 63 cases (13%) in which learners submitted the AI-generated code as their final solution without running the code.

Future studies with more rigorous techniques such as think-aloud or eye-tracking studies are required to properly understand how learners verify and check AI-generated code.

Furthermore, we observed that some learners added code manually to verify the behavior of the AI-generated code on several tasks. For instance, out of the 22 learners who used Codex for a task that required counting the digits of a randomly generated number, the AI-generated code did not include a print statement to display the random number, only displaying the final digits count. Therefore, it made sense for learners to verify the code before submitting it. Among the 11 learners who used Codex by copying the task description, only  $P_{14}$  and  $P_{16}$  correctly added verification code, while four faced difficulties. Furthermore, out of the seven learners who used Codex after manual coding, five correctly added verification code, one struggled, and one did not verify the code at all. Figure 2 displays an example from  $P_{10}$  manually adding verification.

**4.5.3 Modifying AI-Generated Code.** We identified 175 cases in which learners correctly modified the AI-generated code to align it better with the task’s requested behavior. Some edits were towards getting the correct output format. For example, on a task in which learners used Codex which did not produce the correct output format, 10 learners had to modify `message = num1 * num2` into `message = "num1 times num2 is " + str(num1 * num2)`. However, in 57 instances, learners broke correct AI-generated code and were not able to fix it. In some cases, such incorrect edits were accidents (e.g., removing a parenthesis) which left them confused for the rest of the task. In other cases, learners failed to get the desired output format and instead broke the AI-generated code. For example, when  $P_{15}$  tried to add “*Length:*” to the beginning of `print(len(my\_list))`, they instead wrote: `print(len("Length:" + my\_list))`.

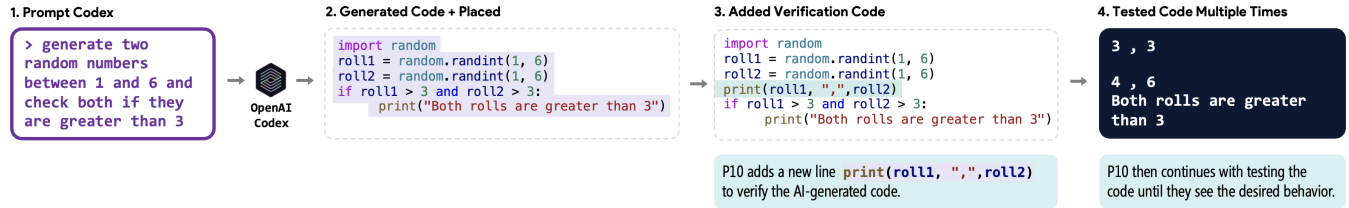


Figure 2: An example of keeping the original code instead of replacing it with AI-generated code ( $P_{12}$ ).

Furthermore, when Codex generated extra, unspecified code, learners had to evaluate whether to delete, keep, or modify the extra code. We analyzed 166 of such instances and found 103 instances (62%) in which learners followed the expected behavior and handled the extra code correctly, while in 50 instances (30%) learners mistakenly retained the extra incorrect code, or in 12 instances (7%) learners deleted the extra useful code.

In 72 cases, Codex generated incorrect code that should have been erased (e.g., when it repeated existing incorrect code). Learners immediately deleted the extra non-useful part of the code in 62% ( $n=45$ ) cases, while keeping the incorrect part in 38% ( $n=27$ ) cases. In extreme cases like when Codex generated only comments for  $P_{12}$ , they executed the code which did not produce any output, but then proceeded and submitted the comments. In a similar situation,  $P_{14}$  had written six lines of code but faced a logical issue they could not fix. They then prompted Codex with the copy of the entire task description. However, Codex repeated their existing incorrect code. Despite this,  $P_{14}$  replaced their original code with the incorrectly repeated code and submitted it without testing instead of deleting the incorrect AI-generated code.

**4.5.4 Using AI-Generated Code as Example.** We identified 26 instances in which learners used the AI-generated code as an example to fix their existing code or write new code similar to the generated code, instead of replacing their code with the newly generated code. For example,  $P_6$  was struggling with concatenating a string and an integer, so they prompted Codex with “message = “num is: “ plus variable num” which generated `message = “num is: “ + str(num)`. However,  $P_6$  did not simply replace their incorrect code with the AI-generated code, but instead carefully compared the two codes and fixed their own code manually. Similarly,  $P_3$  was trying to write an if-else conditional for the first time, however, not knowing the proper syntax, they encountered many issues. Therefore, they prompted Codex with “if else” which generated a sample code for them to fix their code. Similarly, when  $P_2$  was trying to write a for loop to accumulate a number by five for 25 times, they forgot about the colon and indentation which caused an error. To receive help,  $P_2$  prompted Codex with “say hello 10 times” and then fixed their code with the AI-generated code as an example (Figure 3).

## 4.6 RQ2: Effect of AI Code Generator Coding Approaches

In previous sections, we presented results for each Codex usage. Now we shift our focus to how novices used Codex alongside manual programming approaches. Our analysis identified four distinct coding approaches used by learners to incorporate Codex into

their programming practice: **AI Single Prompt**, **AI Step-by-Step**, **Hybrid**, and **Manual**. For each coding approach, we provide a description of the practice, descriptive statistics related to its usage, correlation with code-authoring and code-modification scores during the training phase. See Table 2 for a summary of statistics for each coding approach. We observed a limited number of tasks ( $n=48$ , 3.5%) where learners switched approaches, for which we selected their final approach for our quantitative analyses.

### 4.6.1 AI Code Generator Coding Approaches. AI Single Prompt:

The most frequently used coding approach ( $n=628$ , 46%) was using a single prompt to generate the entire solution (Table 2, Row 1). On 400 tasks, learners prompted Codex with a copy of the task description and submitted the AI-generated code with no manual coding (17 tasks were submitted without testing). Additionally, on 48 tasks, learners wrote the prompt message by themselves using their own words and understanding of the task.

**AI Step-by-Step:** In this coding approach, which was used in 82 tasks (6%), the main body of the submitted code was generated by Codex through multiple and consecutive Codex usages for different parts of the task (Table 2, Row 2). Learners identified multiple subgoals based on the task description or simply broke the task description by its sentences and then prompted Codex with each of the parts.

**Hybrid:** In this coding approach that was used for completing 252 tasks (19%), a few subgoals of the submitted task were generated by Codex while the rest were written manually (Table 2, Row 3). On 57 tasks, learners used Codex after manual coding only to debug their code and fix an encountered issue.

**Manual:** In this coding approach that was used for completing 398 tasks (29%), the final solution submitted by the learners was 100% self-written (Table 2, Row 4). In a special variant of this coding approach that was used in 42 tasks, learners used Codex, but did not use the AI-generated code due to its low-quality.

### 4.6.2 Effect of Coding Approaches on Individual Learning Outcomes.

For every student, we calculated the utilization rate of each coding approach as the number of tasks they used a particular approach divided by the total number of tasks they completed during the training phase. We then calculated a series of Linear Regressions to investigate the relationship between the utilization rate of each coding approach and the immediate and retention post-test scores (on coding tasks and conceptual-MCQs). Our results presented in Figure 4 show the relationship between the utilization of each coding approach and performance on post-test evaluation tests. Although our analysis did not reveal any significant relationships between any of the coding approaches and the post-test evaluation tests, it



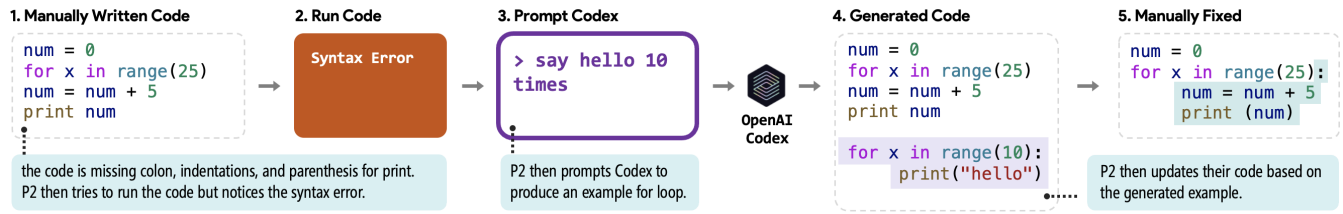


Figure 3: An example of using AI-generated code as an example to fix syntax error with writing loops.

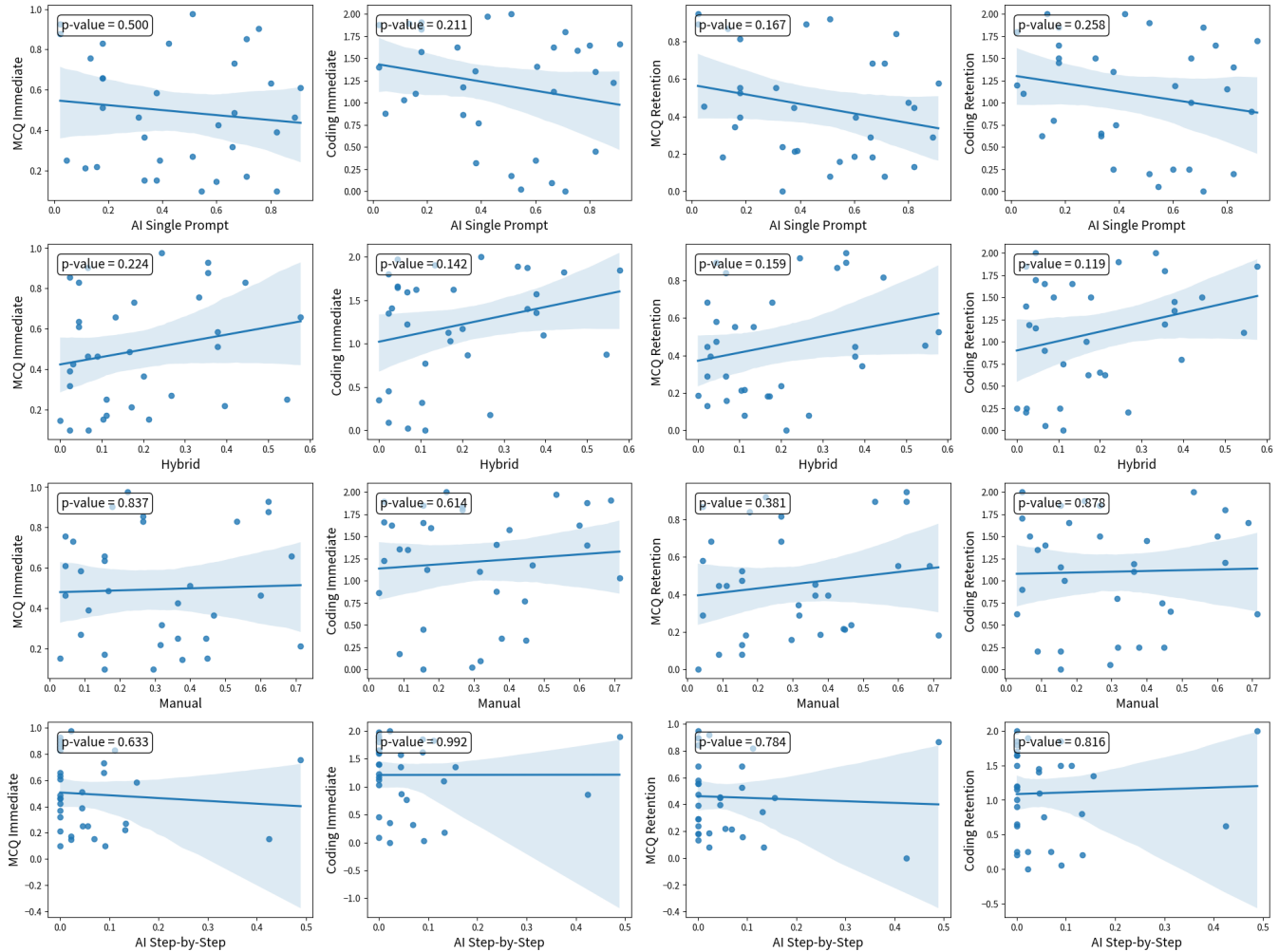


Figure 4: Each row represents correlations between the utilization of a coding approach and each of the four evaluation post-test scores.

shows consistently **positive** trends between the *Hybrid* approach and post-test scores as well as consistently **negative** trends between the *AI Single Prompt* approach and post-test evaluation scores. A possible interpretation of these findings suggests that individuals who actively switched between manual and AI-assisted coding for different parts of a task, demonstrated higher learning outcomes.

However, future, more in-depth studies are required to explore the effect of coding approaches on individual learning outcomes.

**Table 2: Usage count and task correctness score averaged across all tasks in each coding approach.**

| Coding Approach                | Usage | Authoring Score (%)  | Modifying Score (%)  |
|--------------------------------|-------|----------------------|----------------------|
| <i><b>AI Single Prompt</b></i> | 630   | $M=96\%$ , $SD=16\%$ | $M=62\%$ , $SD=45\%$ |
| <i><b>AI Step-by-Step</b></i>  | 82    | $M=76\%$ , $SD=37\%$ | $M=55\%$ , $SD=47\%$ |
| <i><b>Hybrid</b></i>           | 252   | $M=77\%$ , $SD=34\%$ | $M=73\%$ , $SD=39\%$ |
| <i><b>Manual</b></i>           | 397   | $M=63\%$ , $SD=43\%$ | $M=73\%$ , $SD=39\%$ |

## 5 DISCUSSION

### 5.1 Signs of Over-Reliance and Self-Regulation

We have identified multiple signs of self-regulation on AI code generators, where learners executed cognitive control during a learning task [40]. For example, learners actively added code to verify AI-generated code, or tinkered with the AI-generated code to understand the underlying concepts. Additionally, when learners temporarily removed a code that they deemed unnecessary and tested the code to see how it contributed to the program, they were actively engaged in the learning activity and showed signs of self-regulation. Similarly, when learners used AI-generated code as an example to fix their own code, they realized the importance of practice in learning.

However, we also discovered signs of over-reliance [5]. Research in Human-AI collaboration has reported that humans tend to over-trust and rely too heavily on AI agents [38], even when they are incorrect [10, 11]. Furthermore, as a learning support tool, such over-reliance can also connect with ineffective help-seeking behaviors in Interactive Learning Environments (ILE). ILEs are computer-based instructional systems that help beginners learn through task-based environments and support [1, 34]. When students use the on-demand system help, they often rely too heavily on the system’s bottom-out hints that provide the correct answer with little to no explanation [1].

In our analysis, a common sign of over-reliance was the frequent use of the *AI Single Prompt* coding approach, in which learners generated the entire solution using a single prompt. An extreme case of such over-reliance was when learners employed this approach at the beginning, prompted Codex with a copy of the task description, and submitted the generated code without any further editing. This is similar to students’ heavy reliance on bottom-out hints in ILEs which are more likely to harm their learning gains [3, 34]. For example, such over-reliance could result in fake progress during training and difficulties in applying basic concepts later in the training. Another sign of over-reliance was prompting Codex for code similar to existing code. This is related to self-regulation behaviors in help-seeking during problem-solving. Furthermore, this behavior is against the requirement of “reinvestment of received help”, where students are expected to reuse the received help in analogous tasks [42]. Other signs of over-reliance were related to students’ potential over-trust of AI systems. Compared to traditional computer-based programming learning environments that provided correct support [41], AI systems might produce low-quality code that does not lead to the correct solution. Two representative signs of this over-reliance are accepting AI-generated code without verification and misplacement of AI-generated code. This indicates that some

students might believe that AI code generators are fully capable of providing flawless code and they can insert their output into the intended location. Therefore, they assumed that no further action or verification was required by themselves.

Overall, as AI code generators are becoming more prevalent, effective, and accessible through tools like ChatGPT and Copilot, it is important to promote behaviors that contribute to higher learning outcomes while increasing awareness of the dangers of such tools. over-reliance causes learners to process AI information superficially, rather than critically engaging with it using their own knowledge [19]. Therefore, CS-Ed researchers and educators need to be aware of students’ potential misbehavior and integrate best practices of using AI code generators as part of the lesson plan.

### 5.2 Implications for Designers

Although prior research [15, 17, 18] investigated the impact of prompts on the quality of AI-generated code, those studies were mainly conducted offline, without user studies. In contrast, this work provides analyses of how young learners craft prompts, in terms of accuracy, language, and relationship to code. Our analysis also found that poorly crafted prompts led to low-quality code that learners struggled to work with. Therefore, future AI-generation tools could be inspired by *Grounded Abstraction Matching* [33] and support novices in crafting prompts that have a one-to-one relationship with generated code without any missing details, as both a learning activity and programming support tool.

Moreover, to encourage the self-regulated use of AI code generators, such as tinkering, future tools could offer a sandbox for learners to experiment with the prompt until they achieve the desired output through iterative testing and verification. This could increase playfulness as learners may feel less attached to the generated code and be more willing to tinker with it. The sandbox could also be designed in a way to promote tinkering, such as incorporating visual cues that differentiate editable values from fixed syntactical parts of the generated code. These visual cues could be designed in a way to promotes playful tinkering. In addition, properly decomposing problems can support program development processes [30]. However, we found that some students were not effective at task decomposition. Specifically, some students resorted to dividing the task description sentence by sentence for each prompt. Therefore, future design should consider incorporating guided planning components [26] or decomposition charts to train students how to properly decompose a task. LLMs can also help with the decomposition of a task. Future AI code generators targeted at novices could break down a natural language programming prompt into sub-tasks and display them as a Parsons problem. This way

not only the solution is not revealed, but learners are required to actively engage in a puzzle activity before writing code.

## 6 LIMITATIONS

The results presented are based on the participants' uninterrupted interaction with the AI code generator, and do not consider their thought processes and motivations while using Codex. This limitation restricts the depth of our qualitative analysis. Therefore, to fully comprehend the exact motivations behind each Codex usage, what each prompt message meant to the participants, their intentions behind modifying the AI-generated code, and how they verified the code, future introspective and retrospective think-aloud studies are necessary. In addition, our analysis involved only 33 young learners in an online and informal study setting. Other behaviors and usage scenarios may be uncovered with different age groups, as well as in more formal classroom settings like a K-12 or undergraduate programming course. Furthermore, certain behaviors, such as placing the AI-generated code incorrectly, are specific to the way Coding Steps was designed, and therefore may not generalize to other AI code generators.

## 7 CONCLUSION

With the rapid access learners have to Large Language Models (LLMs) through tools like ChatGPT and Github Copilot (based on OpenAI Codex), there are rising concerns about proper learning, over-reliance, and plagiarism. However, these tools are here to stay; therefore, it is necessary to adapt the future of curriculum and tool development based on the most effective strategies and practices that learners use with AI code generators.

This work provides initial insights into how learners use AI code generators, including common usage patterns, the types of prompts they use, and how learners verify and use AI-generated code. Our detailed thematic analysis provide evidence of various signs of over-reliance and self-regulation in learners when interacting with LLM-based code generators. Some learners dealt with AI-generated code properly even when using Codex in "autonomous" modes like the *AI Single Prompt* approach. They manually added code to verify it, and even tinkered with it to better understand the generated code, however, some participants had difficulty integrating the AI-generated code into their solution. In addition, students not only used the original task description to ask for code, but they also crafted prompts with varied language and clarity levels, which sometimes affected the quality of the generated code. Furthermore, we identified four general coding approaches when learners write code with AI code generators: *AI Single Prompt*, *AI Step-by-Step*, *Hybrid*, and *Manual*. We also analyzed the effect of different coding approaches on learning outcomes tested a week later. Our analysis shows consistently **negative** trends between the utilization of the *AI Single Prompt* approach and post-test evaluation scores as well as consistently **positive** trends between the utilization of the *Hybrid* approach and post-test evaluation scores.

## REFERENCES

- [1] Vincent Alevan, Bruce McLaren, Ido Roll, and Kenneth Koedinger. 2006. Toward meta-cognitive tutoring: A model of help seeking with a Cognitive Tutor. *International Journal of Artificial Intelligence in Education* 16, 2 (2006), 101–128.
- [2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [3] Ryan Shaun Baker, Albert T Corbett, and Kenneth R Koedinger. 2004. Detecting student misuse of intelligent tutoring systems. In *Intelligent Tutoring Systems: 7th International Conference, ITS 2004, Maceió, Alagoas, Brazil, August 30-September 3, 2004. Proceedings* 7. Springer, 531–540.
- [4] 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 (2023), 85–111.
- [5] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard-Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 500–506.
- [6] H Russell Bernard, Amber Wutich, and Gery W Ryan. 2016. *Analyzing qualitative data: Systematic approaches*. SAGE publications.
- [7] Andrea J Bingham and Patricia Witkowski. 2021. Deductive and inductive approaches to qualitative data analysis. *Analyzing and interpreting qualitative data: After the interview* (2021), 133–146.
- [8] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1589–1598.
- [9] Jerome Seymour Bruner et al. 1966. *Toward a theory of instruction*. Vol. 59. Harvard University Press.
- [10] Zana Bućinca, Maja Barbara Malaya, and Krzysztof Z Gajos. 2021. To trust or to think: cognitive forcing functions can reduce overreliance on AI in AI-assisted decision-making. *Proceedings of the ACM on Human-Computer Interaction* 5, CSCW1 (2021), 1–21.
- [11] Shiye Cao and Chien-Ming Huang. 2022. Understanding User Reliance on AI in Assisted Decision-Making. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW2 (2022), 1–23.
- [12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [13] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).
- [14] Kathryn Cunningham, Barbara J Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. Avoiding the Turing tarpit: Learning conversational programming by starting from code's purpose. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.
- [15] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 1136–1142.
- [16] Paul Denny, Sami Sarsa, Arto Hellas, and Juho Leinonen. 2022. Robosourcing Educational Resources—Leveraging Large Language Models for Learnersourcing. *arXiv preprint arXiv:2211.04715* (2022).
- [17] 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*. 10–19.
- [18] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A Becker. 2023. My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *Proceedings of the 25th Australasian Computing Education Conference*. 97–104.
- [19] Krzysztof Z Gajos and Lena Mamykina. 2022. Do people engage cognitively with ai? impact of ai assistance on incidental learning. In *27th International Conference on Intelligent User Interfaces*. 794–806.
- [20] Github. 2022. Copilot: Your AI pair programmer. <https://github.com/features/copilot>. [Online; accessed 9-September-2022].
- [21] Kinying Hou, Barbara Jane Ericson, and Xu Wang. 2022. Using Adaptive Parsons Problems to Scaffold Write-Code Problems. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 15–26.
- [22] Michelle Ichinco and Caitlin Kelleher. 2015. Exploring novice programmer example use. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 63–71.
- [23] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.
- [24] Dhanya Jayagopal, Justin Lubin, and Sarah E Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. In *Proceedings*

- of the 35th Annual ACM Symposium on User Interface Software and Technology. 1–15.
- [25] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.
  - [26] Wei Jin, Albert Corbett, Will Lloyd, Lewis Baumstark, and Christine Rolka. 2014. Evaluation of guided-planning and assisted-coding with task relevant dynamic hinting. In *Intelligent Tutoring Systems: 12th International Conference, ITS 2014, Honolulu, HI, USA, June 5–9, 2014. Proceedings 12*. Springer, 318–328.
  - [27] Enkelejd Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* 103 (2023), 102274.
  - [28] 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*. 1–23.
  - [29] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018), 1–43.
  - [30] Kyungbin Kwon and Jongpil Cheon. 2019. Exploring Problem Decomposition and Program Development through Block-Based Programs. *International Journal of Computer Science Education in Schools* 3, 1 (2019), n1.
  - [31] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using large language models to enhance programming error messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 563–569.
  - [32] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
  - [33] 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*. 1–31.
  - [34] Samiha Marwan, Anay Dombe, and Thomas W Price. 2020. Unproductive help-seeking in programming: What it is and how to address it. In *Proceedings of the 2020 ACM conference on innovation and technology in computer science education*. 54–60.
  - [35] Matthew B Miles and A Michael Huberman. 1994. *Qualitative data analysis: An expanded sourcebook*. sage.
  - [36] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. *arXiv preprint arXiv:2210.14306* (2022).
  - [37] Kimberly A Neuendorf. 2017. *The content analysis guidebook*. sage.
  - [38] Samir Passi and Mihaela Vorvoreanu. 2022. Overreliance on AI: Literature review. (2022).
  - [39] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. *arXiv preprint arXiv:2302.04662* (2023).
  - [40] James Prather, Brett A Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What do we think we think we are doing? Metacognition and self-regulation in programming. In *Proceedings of the 2020 ACM conference on international computing education research*. 2–13.
  - [41] Thomas W Price, Rui Zhi, and Tiffany Barnes. 2017. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *Artificial Intelligence in Education: 18th International Conference, AIED 2017, Wuhan, China, June 28–July 1, 2017, Proceedings 18*. Springer, 311–322.
  - [42] Minna Puustinen. 1998. Help-seeking behavior in a problem-solving situation: Development of self-regulation. *European Journal of Psychology of education* 13 (1998), 271–282.
  - [43] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27 (2017), 37–64.
  - [44] 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*. 491–514.
  - [45] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).
  - [46] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 27–43.
  - [47] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 15–26.
  - [48] Renske Smetsers-Weeda and Sjaak Smetsers. 2017. Problem solving and algorithmic development with flowcharts. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*. 25–34.
  - [49] Edward R Sykes. 2010. Design, Development and Evaluation of the Java Intelligent Tutoring System. *Technology, Instruction, Cognition & Learning* 8, 1 (2010).
  - [50] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2023. An Empirical Study of Developer Behaviors for Validating and Repairing AI-Generated Code. Plateau Workshop.
  - [51] 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 *Chi conference on human factors in computing systems extended abstracts*. 1–7.
  - [52] Wengran Wang, Audrey Le Meur, Mahesh Bobbadi, Bitu Akram, Tiffany Barnes, Chris Martens, and Thomas Price. 2022. Exploring Design Choices to Support Novices’ Example Use During Creative Open-Ended Programming. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 619–625.
  - [53] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-side code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.
  - [54] Zhen Xu, Albert D Ritzhaupt, Fengchun Tian, and Karthikeyan Umapathy. 2019. Block-based versus text-based programming environments on novice student learning outcomes: A meta-analysis study. *Computer Science Education* 29, 2-3 (2019), 177–204.