Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language

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

Viraj Kumar
viraj@iisc.ac.in
Indian Institute of Science
Bengaluru, India

Nasser Giacaman
n.giacaman@auckland.ac.nz
University of Auckland
Auckland, New Zealand

ABSTRACT

GitHub Copilot is an artificial intelligence tool for automatically generating source code from natural language problem descriptions. Since June 2022, Copilot has officially been available for free to all students as a plug-in to development environments like Visual Studio Code. Prior work exploring OpenAI Codex, the underlying model that powers Copilot, has shown it performs well on typical CS1 problems thus raising concerns about its potential impact on how introductory programming courses are taught. However, little is known about the types of problems for which Copilot does not perform well, or about the natural language interactions that a student might have with Copilot when resolving errors. We explore these questions by evaluating the performance of Copilot on a publicly available dataset of 166 programming problems. We find that it successfully solves around half of these problems on its very first attempt, and that it solves 60% of the remaining problems using only natural language changes to the problem description. We argue that this type of prompt engineering, which we believe will become a standard interaction between human and Copilot when it initially fails, is a potentially useful learning activity that promotes computational thinking skills, and is likely to change the nature of code writing skill development.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; **CS1**; • **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

OpenAI, GitHub Copilot, foundation models, large language models, CS1, artificial intelligence, introductory programming.

ACM Reference Format:
Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569823

1 INTRODUCTION

Recent breakthroughs in deep learning have led to the emergence of transformer language models that exhibit extraordinary performance at generating novel human-like content such as text (e.g., GPT-3 [5]), images (e.g., DALL-E [20]) and source code (e.g., Codex [6]). Producing source code automatically from natural language prompts promises to greatly improve the efficiency of professional developers [23], and is being actively explored by groups such as OpenAI (Codex), Amazon (CodeWhisperer) and Google (AlphaCode). After less than one year in technical preview, a production version of Codex called Copilot[1] has recently been released as an extension for development environments such as Visual Studio Code. This extension is available for free to students, and claims to be their "AI pair programmer". Just how students will embrace and make use of tools like Copilot is unclear [10], but it seems certain they will play an increasing role inside and outside the classroom.

Very recent work has shown that these code generation models are good at solving simple programming tasks. For instance, Finnie-Ansley et al. evaluated the performance of OpenAI's Codex on a private repository of CS1 exam questions, finding that roughly half of the questions were solved by Codex on its very first attempt [11]. However, little is known about the types of problems for which these models tend to fail, or about how students will interact with code generation tools when such failures occur. One hypothesized interaction that seems very likely is that students will learn to modify, or engineer, natural language problem descriptions to guide the model into generating solutions that "work" (at least in the sense of passing available test cases). Indeed, it is well known that language model outputs are very sensitive to their inputs [21]. For example, when using Codex to solve probability and statistics problems, engineering the prompt to include explicit hints on the *strategy* for solving a problem is extremely effective [25]. An interesting open question in computing education is how well this kind of *prompt engineering* [18] works in practice, and whether it might be a useful activity for students who are learning to program. In this paper, we explore the current performance of Copilot and the efficacy of prompt engineering guided by the following research questions:

RQ1: How well does Copilot perform, at the current time, on a public repository of CS1 programming problems?
RQ2: To what extent do natural language modifications to the problem description lead to the generation of successful solutions after Copilot initially fails?
RQ3: What commonalities exist amongst problems that Copilot cannot solve, despite reasonable prompt engineering effort?

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada.

ACM ISBN 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569823

[1] https://github.com/features/copilot

2 MOTIVATING EXAMPLE

CodeCheck[2] is an online autograder that includes a large collection of publicly accessible programming problems. The goal of CodeCheck is to assist instructors in offering students exercises they can use for self-practice [17]. Consider the following problem which appears as Problem 5 under the "Sums, Averages, Products" category of the "Lists (Simple Exercises)" group in the Python programming problem bank. The problem is displayed below exactly as it appears on the CodeCheck website, with the problem description appearing as a comment within the code editor, the function signature provided, and the statement "Your code here..." as a comment prompting the user to enter their solution:

```python
# Given a list of strings and a string s,
# return the average length of all strings
# containing s.

def averageContainingStringS(strings, s):
    # Your code here...
```

What might happen if a student used Copilot as their "AI pair programmer" here? We provided the problem description to Copilot (in July 2022, shortly after its general availability release), by pasting the comments and function header, excluding the "Your code here..." prompt, into a Visual Studio Code editor with the Copilot extension enabled. The following suggested solution appeared almost instantly:

```python
# Given a list of strings and a string s,
# return the average length of all strings
# containing s.

def averageContainingStringS(strings, s):
    count = 0
    for string in strings:
        if s in string:
            count += 1
    return len(s) * count / len(strings)
```

This is, of course, *incorrect*—the combined length of the matching strings should be divided by their frequency. The arithmetic above is somewhat nonsensical. If the student who used Copilot to generate this solution submitted it to the CodeCheck website for evaluation, they would see that all of the provided test cases fail.

One way this error could be resolved is for the student to fix the arithmetic in the code manually. In this case, Copilot has still served a useful purpose—it helped the student most of the way to a solution by producing code that iterates over the list of strings and that counts the number of matching strings correctly. However a different approach, and one that we envisage as becoming increasingly common, involves modifying the natural language description of the problem and querying Copilot again. How could the prompt be engineered to help guide Copilot towards generating a correct solution? The current (incorrect solution) is not computing the lengths of the matching strings. Consider the following modification to the prompt which adds comment lines 4–6 outlining a simplistic algorithm for solving the problem. With these modifications in place, Copilot produced the following solution:

```python
1  # Given a list of strings and a string s,
2  # return the average length of all strings
3  # containing s.
4  # Count the number of strings that contain s
5  # Also, sum the length of all strings containing s
6  # Finally, divide the total length by the count
7
8  def averageContainingStringS(strings, s):
9      count = 0
10     totalLength = 0
11     for string in strings:
12         if s in string:
13             count += 1
14             totalLength += len(string)
15     return totalLength / count
```

The above solution is correct provided the final value of `count` is positive (the CodeCheck website does not specify, or test for, the return value if none of the strings in `strings` contain `s`). The comments added to lines 4–6 of this example illustrate one potentially valuable pedagogical use of Copilot: *describing computational steps in natural language* as a way of guiding code generation models.

Although prior work in computing education has shown that tools like Codex (which powers Copilot) perform well on typical CS1 problems, little is known about the types of problems for which they tend to fail. In addition, there is currently no work exploring prompt engineering as a strategy for students to resolve errors. We explore both of these ideas using a dataset of publicly accessible problems, thus establishing a baseline for future evaluations of code generation models which we expect will rapidly improve.

3 RELATED WORK

Large language models, or foundation models, are deep neural networks trained with self-supervised learning on broad data sets at a very large scale [4]. These models can then be adapted, or fine-tuned, for application to a wide range of tasks including the generation of natural language, digital images, and source code. While their ability to generate novel human-like outputs is on the one hand fascinating, their rapidly increasing deployment has caused alarm among some researchers and led to calls for better understanding of their implications and risks [3, 24].

GPT-3, released by OpenAI in May 2020, is a groundbreaking large language model that is trained to predict the next token in a text sequence [5]. The Codex model is the result of fine-tuning GPT-3 with an enormous amount of code samples—159GB of code from 54 million GitHub repositories [6]. Copilot is a production version of Codex that has been released as an extension for development environments like Visual Studio Code. It became generally available to all developers in June of 2022, and it is currently free for students and teachers[3]. The impact on educational practice of such technologies is unknown, with arguments on both sides—highlighting concerns of over-reliance by novices [6], and suggesting that the ability to synthesize code automatically could play a revolutionary role in teaching [9, 13].

[2] https://horstmann.com/codecheck

[3] https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers; https://github.blog/2022-09-08-github-copilot-now-available-for-teachers

In the computing education literature, there have been very few evaluations to date of code generation models. Finnie-Ansley et al. explored the performance of Codex on a private dataset of CS1 and CS2 exam problems and on several common variations of the well-known rainfall problem [11, 12]. Codex scored in the 75th percentile when compared to students who were given the same questions, and it was capable of generating multiple correct solutions that varied in both algorithmic approach and code length. As the complexity of problems grow, it is likely that more human interaction with the models will be needed [1]. Sarsa et al. applied Codex to the task of generating novel programming exercises given a single example as input [22]. They found that well over 80% of the generated exercises included a sample code solution that was executable, but that this code passed the test cases that were also generated by Codex only 30% of the time.

Outside of computing education, several recent studies have explored the potential impact of Copilot for developers [8, 15, 19]. Barke et al. observed 20 participants, all of whom had prior programming experience, and found that they were most successful using Copilot when they first decomposed the programming task into microtasks and then prompted Copilot explictly for each of these smaller well-defined tasks [2]. In particular, they observed that almost all of their participants wrote natural language comments as prompts to Copilot, effectively rephrasing the problem description in natural language. A very similar finding was reported by Jiang et al., using a different code generation tool called GenLine, where developers would tend to rewrite the natural language problem specifications in order to clarify their intent to the model [16]. A similar user study involving Copilot was conducted by Vaithilingam et al. to investigate developer perceptions, interaction patterns and coping strategies when the generated code was not correct [26]. They found that when the generated code was incorrect, developers tended to avoid debugging and modifying the code directly, preferring to search for other solutions online or rewrite the code from scratch.

Research suggests that experienced programmers are willing to interact with code generation tools by rewriting problem descriptions in natural language [14]. We expect this type of interaction to become commonplace as code generation tools are widely adopted, and there is evidence that these tools perform best when problem-solving strategies and hints are encoded in the prompts [25]. Learning how to effectively converse with code generation tools will therefore likely be an important skill for novices and conversational programmers [7] to develop in the near future.

4 METHOD

We evaluate the performance of Copilot using Hosrtmann's publicly available CodeCheck exercises. Our evaluation was conducted in July 2022 using the test bank of all 'programming problems' available in Python[4].

4.1 CodeCheck Python Problems

The 166 Python problems are split into 23 sub-categories across four main categories (see Table 1). The top-level categories are:

Table 1: Python exercises from the CodeCheck website

Category	Sub-category	Shortcode	Problem count
Branches	Branches Without Functions	BXF	13
	Branches with Functions	BWF	9
Strings	No Loops	SNL	5
	Comparing Strings	SCS	3
	Finding Substrings	SFS	5
	Words	SW	4
	Numbers in Strings	SNS	3
	Other String Operations	SOO	9
Lists (Simple Exercises)	No loops	LNL	4
	Filling	LF	5
	Maximum and Minimum	LMM	7
	Finding Elements	LFE	6
	Counting Elements	LCE	10
	Sums, Averages, Products	LSAP	14
	Moving or Removing Elements	LMRE	6
	Two Answers	LTA	5
	Double Loops	LDL	8
Two-Dimensional Arrays	No Loops	TNL	8
	Loops Along a Row or Column	TLRC	11
	Looping Over the Entire Array	TLOA	13
	Looping Over Neighbors	TLON	5
	Producing 2D Arrays	TPA	7
	Complex Loops	TCL	6
		Total	**166**

Branches: These 22 problems required some combination of `if/elif/else` statements. The 13 problems in the *Branches Without Functions* sub-category were the only ones across the entire set where CodeCheck did not provide a pre-defined function header. For consistency, we prompted Copilot with "`def`" to generate functions for these problems as well.

Strings: These 29 problems required the use of loops (over the characters of an input string), string slicing, indexing, and basic string methods (e.g., `isdigit()`, `split()`), but without lists or other data structures.

Lists (Simple Exercises): These 65 problems involved searching through lists, counting, averaging, adding/removing/swapping elements, and so on.

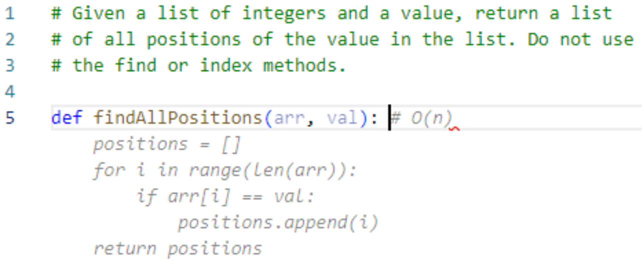
Two-Dimentional Arrays: These 50 problems involved one-dimensional (list) and two-dimensional (list of lists) arrays and required processing some combination of all elements, or corners, borders and diagonals.

4.2 Using Copilot

We copied each problem description from CodeCheck and pasted it into a Visual Studio Code editor with the Copilot extension enabled, as illustrated in Figure 1. The 166 exercises in Table 1 were divided among two of the authors, who used the following protocol:

(1) Copy the problem description (excluding any "*# Your code here...*" comment) and paste it into Visual Studio Code in a blank Python file.
(2) Wait for Copilot to generate a suggestion, and accept it by pressing ↹ (the *tab* key).

[4] https://horstmann.com/codecheck/python-questions.html



```
# Given a list of integers and a value, return a list
# of all positions of the value in the list. Do not use
# the find or index methods.

def findAllPositions(arr, val): # O(n)
    positions = []
    for i in range(len(arr)):
        if arr[i] == val:
            positions.append(i)
    return positions
```

Figure 1: Visual Studio Code editor immediately after pasting a CodeCheck problem. The suggested solution by Copilot appears to the right of the cursor position (to the right of the colon), and can be accepted using the tab key. In this example, the suggested solution begins with an in-line comment showing the computational complexity of the code.

(3) Select and copy the suggested code from the Visual Studio Code editor and paste this into the CodeCheck editor.
(4) Press the CodeCheck button and record the number of test cases that pass and fail.
(5) If all test cases pass, declare the problem "solved" and move to step 1 of the next problem.
(6) If any test cases fail, delete the "buggy" code that was suggested by Copilot.
(7) Observe the failing test cases and engineer the description by adding comments to it that clarify the problem or that provide a strategy for solving the problem. Do not modify any code—only provide natural language descriptions. Repeat steps 6–7 until all test cases pass, or there are no obvious clarifications that can be made to the description.
(8) Record the final engineered problem description, and how many test cases passed as a result of the prompt engineering. Move on to the next problem with step 1.

Step 7 of the protocol was not tightly specified and provided freedom to investigate various prompting strategies. We considered this appropriate given the exploratory nature of this work, and our goal of identifying the types of approaches that lead to success. Nevertheless, this is a limitation that could be addressed in future work. We illustrate typical approaches in Section 5.2.

4.3 Categories of Copilot Failures

Our third research question explores commonalities among the problems that Copilot failed to solve, even after the prompt engineering described in our protocol. To answer this question, the author who did not use Copilot (Section 4.2) independently reviewed the 34 problems on which Copilot failed, and categorized them based on the *original prompt*, the *engineered prompt*, and their own (manual) *solution*. Each category suggests a *possible* cause for failures within that category, but establishing definitive causes is outside the scope of this work.

We categorize 15 problems as *Conceptual* (Table 2), where the original prompt contains one of 6 specific concepts (or terms) that were either retained or slightly reworded in the engineered prompt. Copilot failed on *every* problem (in the full 166-problem set) involving these concepts. For instance, Copilot failed on every problem where the *immediate neighbors* of an element in a string or list have to be swapped (two examples are shown in Figure 5), although it successfully solved problems involving other types of swaps. We hypothesize that our failure to "unpack" these 6 concepts explains Copilot's failure.

Table 2: Copilot failure categories. Subcategory sizes with † denote at least one similar problem that Copilot could solve.

Category	Subcategory	Problem Count
Conceptual	Largest subarray (2), swap neighbors (3), half of odd-length strings (2), 2D arrays with different dim. (2), string prefix (3), centre of 2D array (3)	15
Poor Prompts	Degenerate 2D arrays (2), Other (2)	4
Verbose Prompts	Pattern (2†), Chessboard (1†), Time (2), Position of element in a list (1†), Move/remove elements in a list (3), Longest subsequence (1†), Other (1)	11
Ambiguous	Adjacent duplicates (2†), Other (2)	4

Next, we identified instances where our manual solution required a specific code segment to pass all test cases, but no part of the original or engineered prompts corresponds to this segment (e.g., handling "degenerate" 2D arrays which have just one row). We hypothesize that this lack of correspondence explains Copilot's failure, and we categorize 4 such problems as *Poor Prompts*.

Some of our engineered prompts unpacked concepts in detail (e.g., how to interpret the chessboard position "b8" as row and column numbers). Copilot is based on Codex, which struggles to parse long prompts [6]. We categorize 11 problems as *Verbose Prompts*, where the length of the engineered prompt appears to be the most reasonable explanation for Copilot's failure.

For the remaining 4 problems, we identified ambiguities in the original prompt that allow multiple interpretations. For instance, does reversing the diagonals of a square matrix mean exchanging their elements column-wise, or treating each diagonal as a list to be reversed? In each of these 4 instances, we either failed to address the ambiguity or compounded it in our engineered prompts. We categorize these 4 problems as *Ambiguous*.

Where multiple categories seemed applicable, we preferred the first two categories, since they are backed by evidence: a concept, or a code segment. We provide examples in Section 5.3.

5 RESULTS AND DISCUSSION

5.1 Initial Copilot Performance

Of the 166 problems in the CodeCheck dataset, 79 were solved by Copilot on its first attempt (47.6% success rate). Table 3 summarizes the number of problems in each of the primary categories that were solved successfully or remained unsolved even after modification of the problem description. The 'Verbatim' column tabulates the number of problems that passed all of the test cases when the problem description was initially provided as input to Copilot without any changes. The 'Modified' column shows the number of problems that were successfully solved after manual natural language modification of the problem description.

Table 3: Number of problems that were solved (when the prompt was provided as input verbatim, and after modification) and that remained unsolved after modification.

Category	Verbatim	Modified	Unsolved	Total
Branches	4	13	5	22
Strings	13	7	9	29
Lists	38	19	8	65
Two-D Arrays	24	14	12	50
Total	**79**	**53**	**34**	**166**

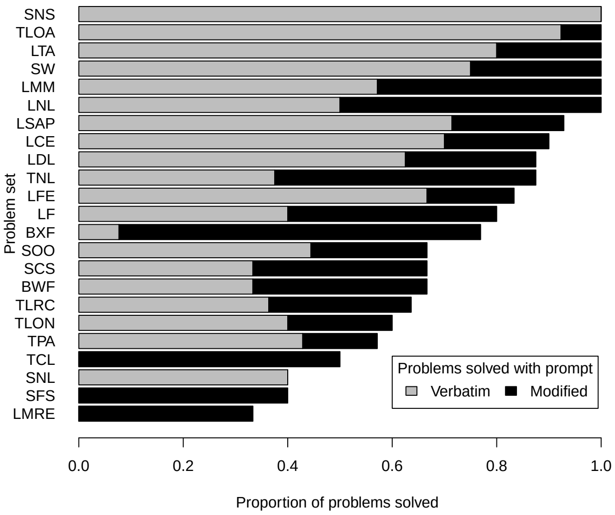


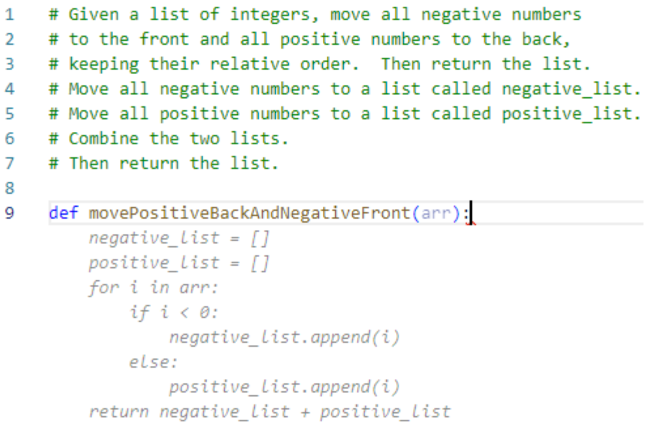
Figure 2: Proportion of problems in each sub-category solved using the problem prompt verbatim, and after modifying the problem prompt as outlined in Section 4. Sub-category codes are detailed in Table 1.

5.2 Effect of Prompt Engineering

A total of 87 problems remained unsolved after the initial code generated by Copilot. Modifying the description for these problems led to Copilot generating a successful solution in 53 cases (60.9%). Figure 2 further breaks down the performance of Copilot into the problem sub-categories. Each row corresponds to one of the problem sub-categories, and illustrates the proportion of problems in that sub-category that were solved successfully without any changes to the problem prompt, alongside the additional proportion that were solved after the prompt was modified.

Performance varied widely across the categories, and although some of this variation is due to categories with relatively small numbers of problems, it is also a result of several categories containing problems with similar solutions. For example, the SNS category contained three problems that required locating numbers in a string (where words and numbers were separated by spaces). The same underlying approach, first splitting the string and then iterating over the items, worked for each problem.

Overall performance was worst for the LMRE category, which involved manipulating list elements. Only two of the six problems in this category were solved by Copilot, and both required prompt engineering. Figure 3 illustrates one of these two problems, with the original description shown on lines 1–3. The initial solution generated by Copilot swapped only the first positive and first negative number. Prompt engineering (see comment lines 4–7) that explicitly suggested building two lists was successful. This example highlights that ‘correctness’ is defined by the test cases, and we declare a problem to be solved if all of the CodeCheck tests pass—that was the case for this solution, even though the value 0 is treated as positive. A more comprehensive set of tests may have necessitated a different solution, and thus more prompt engineering. Overall, CodeCheck problems had an average of 5.1 test cases per problem.



```
1  # Given a list of integers, move all negative numbers
2  # to the front and all positive numbers to the back,
3  # keeping their relative order.  Then return the list.
4  # Move all negative numbers to a list called negative_list.
5  # Move all positive numbers to a list called positive_list.
6  # Combine the two lists.
7  # Then return the list.
8
9  def movePositiveBackAndNegativeFront(arr):
       negative_list = []
       positive_list = []
       for i in arr:
           if i < 0:
               negative_list.append(i)
           else:
               positive_list.append(i)
       return negative_list + positive_list
```

Figure 3: Suggested solution (correct after prompt engineering) to problem 4 in the LMRE category. The manually added comments appear on lines 4–7.

Prompt engineering was successful to some extent in every category where there was an initial failure with the exception of SNL. The three problems in this category which consistently failed, despite prompt engineering, involved arbitrary string manipulations and specific special cases. For example, the full prompt for problem 5 in this category was as follows: *“# Given a string, return the string with the first half and the second half doubled. For example, Java becomes JaJavava and Hello becomes HelHellolo. If the length is odd, like Hello, consider the middle character as part of the first half. If the string is less than 2 characters long, return the original string”.*

In general, when prompt engineering was required, modifications that resembled pseudocode tended to be the most successful. For example, consider the original description for problem 6 in the TLOA category, which was the only problem in this category that required prompt engineering: *“#Given a two-dimensional array of integers, shift each row by one to the right and put a 0 at the leftmost column. The rightmost column is lost. Then return the updated array”.* Copilot tended to wrap the rightmost element into the leftmost position, rather than inserting 0. The following modified prompt, which is more explicit about the insertion of 0, yielded a correct solution: *“#For each row: 1) remove the value at the rightmost position. 2) insert the value 0 into index position 0”.*

```
# Given integers n and k, return a list
# of length n containing 0 1 2 ... k-1 0 1 2 ... k-1 ...
# n times.
# First generate a list from 0 to k-1, then repeat it n times.
def repeatSequence(n, k):
```

```
# Given an integer n, return a list containing
# 1 2 2 3 3 3 4 4 4 4 ... and finally n repeated n times.
# You may assume n is 0 or greater.
# The list should begin with the value 1
# Then contain the value 2 twice
# Then contain the value 3 three times, and so on
# Finally, the list should end with the value n repeated n times.
def reapeatNumTimes(n):
```

Figure 4: Copilot solved the upper problem using a terse additional prompt (shown in green), but could not solve the similar lower problem in the *Verbose Prompts* (*Pattern*) subcategory. Note the verbose additional prompt (shown in red).

5.3 When Prompt Engineering Fails

In our study, the two largest categories of problems on which Copilot failed (Table 2) are *Conceptual* and *Verbose Prompts*. We provide illustrations of each type. The original prompt for the upper problem in Figure 4 is poor: it explicitly asks for a list of length n, whereas the test cases expect a list of length k*n. The terse additional prompt (shown in green) helps Copilot produce the correct answer (despite retaining the original prompt). In contrast, the original prompt for the lower problem in Figure 4 seems unambiguous. Both problems require Copilot to inductively infer a pattern, but Copilot fails on the lower problem, despite the additional chain of instructions (shown in red). Chen et al. acknowledge an exponential drop in the model's performance as these chains grow in length [6]. Hence, we categorize this problem as *Verbose Prompts* (*Pattern*).

Figure 5 shows two similar problems from the *swap neighbors* subcategory within the *Conceptual* category. The original prompt of the upper problem specifies that c cannot be the first or last character of s. The lower problem could arguably belong to the *Poor Prompts* category, because neither the original nor the engineered prompt specifies this restriction. Further, for the string acbcd, neither prompt recognizes that the result could be bcdca or dcacb depending on the order in which the swaps around c occur. However, code that resolves these ambiguities is unnecessary since none of CodeCheck's tests examine such inputs. Since the poor prompt does not account for Copilot's failure, we categorize this problem as *Conceptual*. In both cases, the engineered prompts are also verbose. This illustrates an interesting tension between the need to unpack concepts while keeping the prompt short. It may be possible to engineer prompts more effectively by rewriting the original prompts, rather than by adding to them as we have done.

6 LIMITATIONS AND FUTURE WORK

All problems in this study were in Python, and procedural in nature without utilizing its object-oriented or functional capabilities. Exploring the performance of Copilot on more complex problems, as well as the use of different prompt engineering strategies, would be valuable to improve our understanding of the use of code generation models in educational contexts. The primary focus of our study was on the prompt engineering required to "converse with Copilot". Observing novice programmers and their authentic interactions with Copilot would be a fascinating avenue for future work.

```
# Given a string s and a character c, return a string with the characters
# surrounding the first of c reversed. For example, if s is Hello-World
# and c is -, return HellW-oorld. You may assume that the character in c
# is not the first or last character in the string.
# Find the position of the character c in the string s
# Find the character to the left of that position
# Find the character to the right of that position
# Swap the two characters
# Only those two characters should change.
# The output string should be the same length as the input string.
def swapAroundGivenCharacter(s, c):
```

```
# Given a string s and a character c, return a string with the characters
# surrounding any occurrence of c reversed. For example, if s is 'Hello
# beautiful world' and c is a space, return 'Hellb oeautifuw lorld'. Skip
# any positions where c occurs twice in a row.
# That is, for every occurrence of character c:
# 1) get the character on the left of c
# 2) get the character on the right of c
# 3) swap those two characters
def reverseAroundC(s, c):
```

Figure 5: Two problems in the subcategory *Conceptual* (*swap neighbors*). The engineered prompts resemble pseudocode and are shown in red.

Codex, the model that underlies Copilot, is non-deterministic [6] which introduces challenges around the replicability of our results. Anecdotally, when returning to some of the problems in our dataset after several weeks, we observed different code suggestions than were originally produced. Nevertheless, across the 166 problems we evaluated, overall we observe similar rates of initial success reported in earlier studies [11]. As we expect code generation models like Codex to rapidly improve over time, we present our results on a public dataset as a current baseline.

We see numerous future work opportunities in exploring the impact that Copilot will have on programming teaching, learning, and assessment. Open questions remain on the ethics of students using Copilot to complete assignments. Is it considered academic misconduct for students to incorporate code suggested by Copilot, or is this merely considered an IDE auto-complete feature? How will code similarity tools fare in detecting code written exclusively by Copilot? In what ways can Copilot be embraced as a valuable learning tool to help students improve their computational thinking skills? How will introductory programming courses adapt to the growing use by students of such tools, and how can strategies for constructing effective input prompts be explicitly taught?

7 CONCLUSION

Generative language models look set to radically change the way that computing courses are taught and the way that students learn to program. However, such models are very sensitive to their input prompts, and the ability to engineer effective prompts that generate correct solutions will be an important interaction skill for students in the future. We present the first exploration of the efficacy of prompt engineering for Copilot in an introductory programming context. Roughly half of the problems were solved using the original problem descriptions verbatim, and more than half of the remaining problems were solved by engineering the prompts to contain explicit algorithmic hints, which was effective across almost all categories of problems. We see pedagogical value in these interactions with Copilot, as students need to reflect on code failures and translate the abstract concepts contained in problem descriptions into concrete computational objects and steps, and then express these in natural language.

REFERENCES

[1] 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).

[2] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2022. Grounded Copilot: How Programmers Interact with Code-Generating Models. https://doi.org/10.48550/ARXIV.2206.15000

[3] 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* (Virtual Event, Canada) *(FAccT '21)*. Association for Computing Machinery, New York, NY, USA, 610–623. https://doi.org/10.1145/3442188.3445922

[4] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. 2021. On the Opportunities and Risks of Foundation Models. https://doi.org/10.48550/ARXIV.2108.07258

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[7] 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* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 15 pages. https://doi.org/10.1145/3411764.3445571

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

[9] Paul Denny, Sami Sarsa, Arto Hellas, and Juho Leinonen. 2022. Robosourcing Educational Resources – Leveraging Large Language Models for Learnersourcing. https://doi.org/10.48550/ARXIV.2211.04715

[10] Neil A Ernst and Gabriele Bavota. 2022. AI-Driven Development Is Here: Should You Worry? *IEEE Software* 39, 2 (2022), 106–110.

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

[12] 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 *Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE '23)*. Association for Computing Machinery, New York, NY, USA, 10 pages. https://doi.org/10.1145/3576123.3576134

[13] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (Hagenberg, Austria) *(PPDP '10)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1836089.1836091

[14] Geert Heyman, Rafael Huysegems, Pascal Justen, and Tom Van Cutsem. 2021. Natural language-guided programming. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 39–55.

[15] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 319–321. https://doi.org/10.1109/ICSE-Companion55297.2022.9793778

[16] 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* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 386, 19 pages. https://doi.org/10.1145/3491102.3501870

[17] Deepak Kumar. 2018. REFLECTIONS Tools from the Education Industry. *ACM Inroads* 9, 3 (aug 2018), 22–24. https://doi.org/10.1145/3233246

[18] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586* (2021).

[19] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 1–5. https://doi.org/10.1145/3524842.3528470

[20] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical Text-Conditional Image Generation with CLIP Latents. https://doi.org/10.48550/ARXIV.2204.06125

[21] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. https://doi.org/10.48550/ARXIV.2102.07350

[22] 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 18th ACM Conference on International Computing Education Research* (Lugano, Switzerland) *(ICER 2022)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3501385.3543957

[23] Jiho Shin and Jaechang Nam. 2021. A Survey of Automatic Code Generation from Natural Language. *Journal of Information Processing Systems* 17, 3 (2021), 537–555.

[24] Alex Tamkin, Miles Brundage, Jack Clark, and Deep Ganguli. 2021. Understanding the Capabilities, Limitations, and Societal Impact of Large Language Models. https://doi.org/10.48550/ARXIV.2102.02503

[25] Leonard Tang, Elizabeth Ke, Nikhil Singh, Bo Feng, Derek Austin, Nakul Verma, and Iddo Drori. 2022. Solving Probability and Statistics Problems by Probabilistic Program Synthesis at Human Level and Predicting Solvability. In *Artificial Intelligence in Education. Posters and Late Breaking Results, Workshops and Tutorials, Industry and Innovation Tracks, Practitioners' and Doctoral Consortium*, Maria Mercedes Rodrigo, Noburu Matsuda, Alexandra I. Cristea, and Vania Dimitrova (Eds.). Springer International Publishing, Cham, 612–615.

[26] 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* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665