The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming

James Finnie-Ansley
The University of Auckland
Auckland, New Zealand
james.finnie-ansley@auckland.ac.nz

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

Brett A. Becker
University College Dublin
Dublin, Ireland
brett.becker@ucd.ie

Andrew Luxton-Reilly
The University of Auckland
Auckland, New Zealand
a.luxton-reilly@auckland.ac.nz

James Prather
Abilene Christian University
Abilene, Texas, USA
james.prather@acu.edu

Mary has been collecting daily rainfall data and has stored the data in a list. She wants to calculate the average of the rainfall data (excluding any negative values) and the maximum. Write a function that returns these two values as a tuple, or -1 if there is no valid data.

```
def rainfall(data):
    total = 0
    count = 0
    max_rain = 0
    for i in range(len(data)):
        if data[i] >= 0:
            total += data[i]
            count += 1
            if data[i] > max_rain:
                max_rain = data[i]
    if count == 0:
        return -1
    else:
        return total / count, max_rain
```

```
def rainfall_data(rainfall):
    valid_data = []
    for i in rainfall:
        if i > 0:
            valid_data.append(i)
    if len(valid_data) == 0:
        return -1
    else:
        return (sum(valid_data)/len(valid_data),
                max(valid_data))
```

Figure 1: A problem description (left) based on the Rainfall Problem, provided verbatim to Codex as input, and two different programs generated (output) by Codex (center and right). Both output programs meet the requirements of the problem.

ABSTRACT

Recent advances in artificial intelligence have been driven by an exponential growth in digitised data. Natural language processing, in particular, has been transformed by machine learning models such as OpenAI's GPT-3 which generates human-like text so realistic that its developers have warned of the dangers of its misuse. In recent months OpenAI released Codex, a new deep learning model trained on Python code from more than 50 million GitHub repositories. Provided with a natural language description of a programming problem as input, Codex generates solution code as output. It can also explain (in English) input code, translate code between programming languages, and more. In this work, we explore how Codex performs on typical introductory programming problems. We report its performance on real questions taken from introductory programming exams and compare it to results from students who took these same exams under normal conditions, demonstrating that Codex outscores most students. We then explore how Codex handles subtle variations in problem wording using several published variants of the well-known "Rainfall Problem" along with one unpublished variant we have used in our teaching. We find the model passes many test cases for all variants. We also explore how much variation there is in the Codex generated solutions, observing that an identical input prompt frequently leads to very different solutions in terms of algorithmic approach and code length. Finally, we discuss the implications that such technology will have for computing education as it continues to evolve, including both challenges and opportunities.

CCS CONCEPTS

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

KEYWORDS

academic integrity; AI; artificial intelligence; code generation; code writing; Codex; copilot; CS1; deep learning; introductory programming; GitHub; GPT-3; machine learning; neural networks; novice programming; OpenAI

ACE '22, February 14–18, 2022, Virtual Event, Australia

ACM ISBN 978-1-4503-9643-1/22/02.
https://doi.org/10.1145/3511861.3511863

ACM Reference Format:
James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conference (ACE '22), February 14–18, 2022, Virtual Event, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3511861.3511863

1 INTRODUCTION

On September 10, 2021 the *New York Times* ran an article titled "A.I. Can Now Write Its Own Computer Code. That's Good News for Humans" describing OpenAI's[1] Codex AI model at a very high level [27]. Codex is a descendant of GPT-3[2] one of the most advanced natural language models available. Codex is trained on more than 50 million GitHub repositories representing the vast majority of the Python code available on GitHub. Codex can take English-language prompts and generate code in several programming languages. It can also translate code between programming languages, explain (in English) the functionality of code provided as input, and return the complexity of code it generates. It also has the ability to utilise APIs allowing it to, for example, send emails and access information in databases. Codex is available via the OpenAI API[3] and it also powers GitHub Copilot[4] which is billed as "Your AI pair programmer" – a direct reference to pair programming, an approach well-known in computing education research and practice [26] which was a hallmark of the Extreme Programming (XP) software development methodology [41].

The NYT article provided several examples of Codex output including "make a snowstorm on a black background" to which Codex provides JavaScript code to draw a black rectangle with randomly – yet appropriately – sized and placed white shapes which then move downward, at an expected speed. Codex was also discussed on a September 2021 Lex Fridman podcast, during an interview with Donald Knuth [12]. Fridman stated that "This puts the human in the seat of fixing issues versus writing from scratch". Knuth was overall sceptical about AI generated code, lack of control when systems are built on auto-generated code, and executing code with function that is not fully understood, stating "I'm never going to try to write a book about that... I'm on the side of understanding".

1.1 Motivating Example

The English text in Figure 1 shows a prompt that we provided to Codex. This is a unique variant of Soloway's Rainfall Problem [39] which has been used in a number of studies of programming ability over the past 30+ years and which has developed a reputation for being surprisingly difficult for introductory-level (CS1 [3]) students [32] with multiple studies confirming poor performance [14]. This variant is fairly standard, but the English description contains extraneous information such as "Mary has been collecting daily" and abstract requirements/references such as "rainfall data" and "these two values". It also contains very specific requirements such as "returns these two values as a tuple", but also very human-like requirements such as "excluding any negative values" (in brackets). The two functions shown in Figure 1 (center and right) were generated by Codex in separate runs, in response to the same input prompt (left). Both are functionally correct, well-structured, variables are well-named, and potential barriers such as extraneous information and abstract requirements do not seem to cause issues.

This technology should be of great interest to all computing educators. What we are dealing with is a freely-available program that can take casually defined English language problem specifications, much like typical exam questions, and return often-correct, well-structured code that could pass as human-written. It can also translate code into other languages, return complexity data with code solutions and provide a complete program or a standalone function to complete a specified task by stating either "write a program to..." or "write a function to..." in the input. It is also quite fast – we never waited more than a second or so for a web-based API output in any of the experiments we describe in this paper.

The potential implications of Codex for computing education are significant, but the effectiveness of Codex in introductory computing contexts is unknown. In this paper we explore how Codex performs on typical introductory programming exercises, compare its performance to that of real students, explore the variations in Codex generated solutions, and explore the resulting implications for the future of computing education. Our research questions are:

RQ1: How does Codex perform on first year assessments compared with CS1 students?

RQ2: How does Codex perform on variations of a benchmark computing education problem that differ in context and level of detail?

RQ3: How much variety is there in the solutions generated by Codex?

The remainder of this paper is organised as follows: Section 2 provides the background of Codex and focuses on literature involving GPT-3 and OpenAI. We then turn to an evaluation of Codex for introductory programming problems with Section 3 reporting our method and Section 4 reporting the effectiveness of Codex in generating code solutions to real CS1 exam questions and solving variants of the Rainfall Problem [39]. We discuss the implications of our findings further in Section 5.

2 RELATED WORK

The concept of computer programs being generated by computer programs goes back several decades [23], but has only gained significant ground very recently. A comprehensive review of introductory programming literature from 2018 makes no mention of tools that use AI to produce code, and only mentions a few machine learning based systems to analyse code or provide feedback along with intelligent tutoring systems that aim to provide customised learning approaches (normally, pace and topic choice) for learners [22]. One prerequisite for AI-generated code was the availability of training data which has been growing for at least two decades. However, it wasn't until the AI/ML "boom" in 2017 [13] that natural language models and other applications of AI began to make strong progress.

In 2020, OpenAI released GPT-3 [4], their 3rd generation Generative Pre-trained Transformer, a deep-learning natural language prediction model. GPT-3 has 175B parameters and was trained on 570GB of text [40]. GPT-3 has an "unusually large" capability set including text summarization, chatbot behaviour, search, and essay generation [4]. The improvements that GPT-3 made over GPT-2, a functionally similar model trained on a dataset approximately 10% as large, were described by experts as remarkable given that they are the result of scaling model and training data size. Those

[1] OpenAI (openai.com) is a non-profit "AI research and deployment company" [28] set up with a $1 billion pledge from a group of founders including Tesla CEO Elon Musk [33]. Microsoft has heavily funded OpenAI and now licenses GPT-3 exclusively [31]. It is considered a competitor to Alphabet's DeepMind [11].
[2] GPT-3 stands for third-generation Generative Pre-trained Transformer
[3] beta.openai.com
[4] copilot.github.com

experts, a group of researchers from OpenAI, the Stanford Institute for Human-Centered Artificial Intelligence, and other universities, convened under Chatham House Rules in 10/2020 to discuss the capabilities, limitations, and societal impact of GPT-3 [40].

Despite not being trained for the generation of computer programs, testing of GPT-3 revealed that it could generate rudimentary Python programs when supplied with Python docstrings [4]. In 2021, OpenAI released Codex, a descendent of GPT-3 that was trained on an additional 159GB of Python code from >50M GitHub repositories [5]. Codex is "proficient" in over a dozen programming languages including JavaScript, Go, Perl, PHP, Ruby, Swift, TypeScript, and Shell, but is "most capable" in Python [42] which is unsurprising as that is what it was trained on. However, this makes the fact that it is proficient in other languages particularly exciting.

The OpenAI team released a paper on arXiv on July 14, 2021 [5] presenting Codex and their initial testing. This paper measured the functional correctness of Codex in synthesising programs from docstrings. Repeatedly sampling from the model was shown to be particularly effective in producing working solutions to 164 "difficult" problems. These problems were released as a dataset in the same paper and hand written because Codex was trained on a large fraction of GitHub containing the solutions to many problems from a large number of sources. Codex produced functionally correct programs for 70% of the prompts. The testing also revealed limitations including that Codex struggles to deal with longer and higher-level specifications. A specific weakness noted was binding operations to variables, similar to text-conditional generative models often struggle with binding attributes to objects [5]. The authors also described some broader impacts and conducted a hazard analysis. Although the authors claim that Codex and similar technologies may "aid in education" it is likely that many educators may have more serious concerns when Codex is in the hands of students.

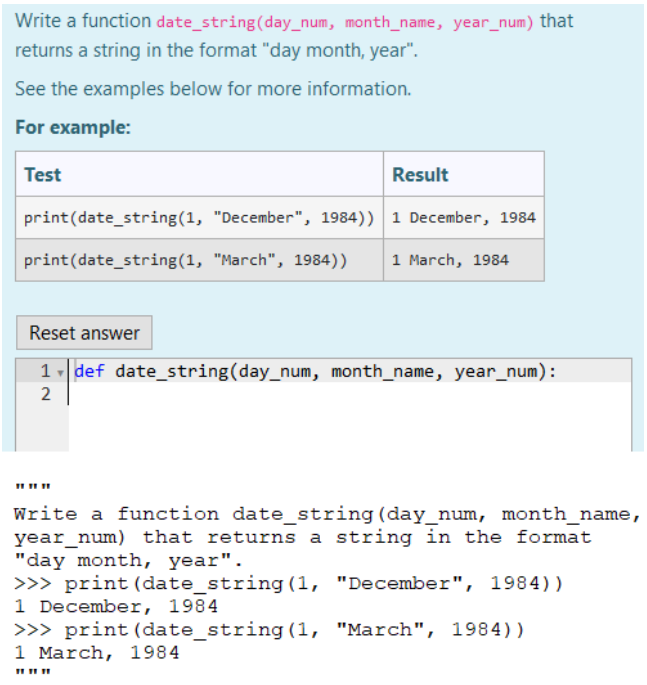

Figure 2: Presentation of the first question of Test 1 within the web-based examination tool as seen by students (top, shaded background); and the format of the same question as provided to Codex (bottom, between """).

3 METHOD

We assess the accuracy of Codex when applied to CS1-type problems using two evaluations. The first involves 23 programming questions that were used as summative assessments on two invigilated lab-based tests conducted in the CS1 programming course at our institution in 2020. In this evaluation, we provide as input to Codex the problem statements exactly as they were presented to students. The second evaluation involves six variations of the problem wording for the classic Rainfall Problem [39] published in the literature and one variation we have used in our teaching. In this evaluation, we provide as input to Codex the problem statements exactly as they appeared in the literature and as used in our teaching.

All runs were on the "Davinci" Codex model (the most capable, but slowest). We use a *temperature* of 0.9 (or 90%) for all runs. Higher temperature values produce more random responses, and the Codex documentation suggests a temperature of 0.9 for "more creative applications". Since we were interested in creative answers and the diversity of Codex's solutions, we chose to follow this suggestion.

3.1 CS1 Programming Tests

In this evaluation, we use questions that appeared on two CS1 programming tests conducted at our institution in 2020. These tests were the primary invigilated assessments in the course, with Test 1 conducted near the middle of the course and Test 2 conducted at the end. Both tests were conducted under examination conditions (timed, invigilated) using lab-based software that automatically graded student solutions. The image at the top of Figure 2 illustrates how the first question of Test 1 appeared to students. The problem statement is listed first, followed by several example test cases. Underneath these is a code editor into which students typed their solutions. Upon submission, students received immediate feedback, and were shown the first failing test case. Incorrect submissions attracted a penalty of 5% applied to that question, which accumulated over subsequent submissions up to a maximum penalty of 50%. Marks for each question were assigned on an all-or-nothing basis: marks were only awarded for a question (including any penalties) if the submitted code successfully passed all of the tests. Code that did not successfully pass all tests for a question received 0 marks. Test 1 contained 11 questions and Test 2 contained 12 questions. Questions were ordered with respect to (approximate) increasing complexity on each test. Table 1 provides the wording of three problems (Q1, Q5, Q11) taken from near the start, middle and end of Test 2, to illustrate the typical wording used.

We evaluate the performance of Codex on the 23 questions from these two tests. The image at the bottom of Figure 2 illustrates how each problem was reformatted for presentation to Codex. Input to Codex is in the form of a Python docstring (within pairs of three double quotes: """). For each question, the problem statement

Table 1: Several examples of questions from Test 2, illustrating typical language used in problem prompts. All of these examples were solved correctly by Codex.

Q1: Write a function named `count_odd(my_list)` that returns the number of odd integers in a given list.
Q5: Write the `get_numbers_needed()` function which takes two parameters: a list of numbers `(numbers)` and an integer `(target)` . The function returns a list of all the numbers from the parameter list (starting from the beginning) which add up to exactly, or just over, the target parameter, e.g. if the target is 21 and the parameter list is [15, 10, 5, 20], the function returns the list made up of the first two numbers, [15, 10] , which have a sum of 25. If the sum of the parameter list numbers is less than the target parameter, the function returns the empty list.
Q11: Complete the function `sort_contact_tuple(contact_tuple)` that takes a tuple `contact_tuple` as a parameter. The tuple contains the extension details of staff members at a business and is formatted as follows: (name1, extension1, name2, extension2,...). You can assume that the tuple always has an even number of entries and that each name in the tuple is unique. The `sort_contact_tuple()` function will sort the contact details based on the staff names in ascending alphabetical order (A to Z). It will then return a tuple with the sorted details.

was used verbatim, and any example test cases were included by showing the expected output for a given input.

For each question, we took the response generated by Codex and executed it using the same test cases as were applied during the invigilated tests. This was achieved by submitting the Codex response as input to the examination software. If a response successfully passed all of the tests, we moved on to the next question. If a response did not pass all of the tests for a question, we generated a new response by resubmitting the problem statement to Codex. We generated a maximum of 10 responses from Codex for any question, at which point we abandoned the question and considered it unsolved. This "repeated sampling" approach was used by the developers of Codex in their testing [5] however our abandonment threshold (10 attempts) was much lower than theirs (164). We recorded the total number of submissions required to solve each question. In a small number of cases, which we document in our results, if the response from Codex was correct up to a trivial formatting error (for example, a missing comma in the expected output) we manually amended the response and counted this as an extra submission. This is consistent with the approach a student might take if they submit code which is algorithmically correct, but includes a minor formatting error. The examination software we use highlights such formatting errors to students.

3.2 Rainfall Problem Variants

In this evaluation, we use seven variations of the well-known Rainfall Problem [39], six of which are published in the literature. The seventh variation was used in a CS1 course at our large, research intensive university in New Zealand. This variant uses harvested apples rather than rainfall as the contextual setting and so will be referred to as the *apples* variant. Table 2 shows the wording of these seven variants exactly as they were provided to Codex. In this case, no example test cases were provided as part of the problem

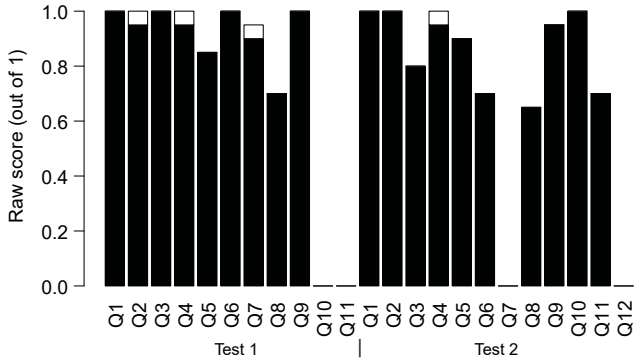

Figure 3: Raw score achieved by Codex on CS1 test problems (accumulating penalties applied for incorrect submissions; problems abandoned after 10 failing submissions). Empty caps on some bars indicate potential scores in the absence of trivial errors.

description. The problem descriptions are taken verbatim from the corresponding source articles, with one exception. In the article by Simon [37], the problem description includes a graphical figure representing one possible input array; we have omitted the image as there is no support for providing diagrams as input to Codex.

Each problem description was provided to Codex in a docstring 50 times. For the Fisler [10], Simon [37], Guzdial et al. [16], Lakanen et al. [18], and *apples* problems, a function header was also provided as the problem descriptions prompt the solver to write a function or imply lists are provided as arguments rather than standard input. Each response was executed against 10 test cases we prepared. Thus, we evaluated a total of 350 responses, each against 10 test cases, for a total of 3500 evaluations. We recorded how many of these tests pass for each response as well as high-level metrics to evaluate solution structure such as the classic "one loop or two?" distinction commonly debated regarding the Rainfall Problem [10].

4 RESULTS

4.1 CS1 Programming Tests

Our first research question asks "How does Codex perform on first year assessments compared with CS1 students?". Figure 3 shows the outcome of the responses generated by Codex for the 23 programming questions from Tests 1 and 2 of our CS1 course. Of the 23 questions, all but 4 were solved successfully using fewer than 10 responses, and 4 responses (for Q2, Q4 and Q7 of Test 1 and Q4 of Test 2) generated the correct solution with the exception of a trivial formatting error. Overall, nearly half of the questions (10) were solved successfully on the first attempt (including solutions with a trivial formatting error).

4.1.1 Comparison With Student Performance. To contextualise this performance, we calculate the score that the responses generated by Codex would have received if graded according to the question weights and accumulated penalties used for Tests 1 and 2 with real students. Test 1 was graded out of a total of 20 marks, with Q1 and Q11 worth 1 mark and all other questions worth 2 marks. Test 2 was graded out of a total of 25 marks, with Q11 and Q12 worth 1 mark, Q4, Q9 and Q10 worth 3 marks, and all other questions

Table 2: Variations of the wording of the "Rainfall" problem.

Reference	Problem wording
Soloway [39]	Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.
Ebrahimi [9]	Write a program that will read the amount of rainfall for each day. A negative value of rainfall should be rejected, since this is invalid and inadmissible. The program should print out the number of valid recorded days, the number of rainy days, the rainfall over the period, and the maximum amount of rain that fell on any one day. Use a sentinel value of 9999 to terminate the program.
Simon [37]	A program has a one-dimensional array of integers called iRainfall, which is used to record the rainfall each day. For example, if iRainfall[0] is 15 and iRainfall[1] is 0, there was 15mm of rain on the first day and no rain on the second day. Negative rainfall values are data entry errors, and should be ignored. A rainfall value of 9999 is used to indicate that no more rainfall figures have been registered beyond that element of the array; the last actual rainfall value recorded is in the element immediately before the 9999. The number of days represented in the array is open-ended: it might be just a few days, or even none; it might be a month; it might be several years. The number of days is determined solely by the location in the array of the 9999 entry. Write a function method to find and return the average rainfall over all the days represented in the array. A day with negative rainfall is still counted as a day, but with a rainfall of zero.
Fisler [10]	Design a program called rainfall that consumes a list of numbers representing daily rainfall amounts as entered by a user. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list.
Guzdial et al. [16], cited in [15]	Write a function rainfall that will input a list of numbers, some positive and some negative, e.g., [12, 0, 41, -3, 5, -1, 999, 17]. These are amounts of rainfall. Negative numbers are clearly a mistake. Print the average of the positive numbers in the list. (Hint: The average is the total of the positive numbers divided by the number of just the positive numbers.)
Lakanen et al. [18]	Implement the 'Average' function, which takes the amounts of rainfall as an array and returns the average of the array. Notice that if the value of an element is less than or equal to 0 ('lowerLimit'), it is discarded, and if it is greater than or equal to 999 ('sentinel'), stop iterating (the sentinel value is not counted in the average) and return the average of counted values.
apples	Create a method called harvest that takes one parameter that is a list of integers representing daily tonnes of fruit picked at a given orchard. It returns a floating point number rounded to 1 decimal place representing an average of the non-negative amounts up to either the first sentinel or the end of the list, whichever comes first. The sentinel is -999. If it is not possible to compute an average, then return -1.0. It is not possible to compute an average if there is no valid list (i.e. the parameter is None), or there are no non-negative values before the sentinel. There may be values after the sentinel but they are to be ignored when determining the average.

worth 2 marks. Taking into account the penalty scheme, where each incorrect submission attracts a 5% penalty applied to the final mark for the corresponding question, the Codex responses scored a total of 15.7/20 (78.5%) for Test 1 and 19.5/25 (78.0%) for Test 2. Figure 4 plots the scores (scaled to a maximum of 100) of 71 students enrolled in the CS1 course in 2020 who completed both tests. The performance of the responses generated by Codex is marked with a red asterisk. Averaging both Test 1 and Test 2 performance, Codex's score is in position 17 when ranked alongside the 71 students' scores, placing it within the top quartile of class performance.

4.1.2 Trivial Formatting Errors. The Codex generated response produced incorrectly formatted output for three of the questions on Test 1 (Q2, Q4, Q7) and one question on Test 2 (Q4). For example, a test case in Test 1 Q2 prompted students to print *"The value of 4x is 8."* whereas the response generated by Codex, while otherwise correct, excluded the full stop. Similarly, a full stop was missing in the output for Test 1 Q4 as well as an incorrectly worded prompt. This question asked for a program that prompted the user to enter a positive integer, and the example test cases displayed this prompt as: *"Enter a positive integer:"*, whereas the prompt generated by the Codex response was: *"Enter a number:"*

The error for Q4 of Test 2 also involved the printing of a prompt for the user. This question asked for a program that would repeatedly read an input string from the user until a string that met certain conditions was received. The example test cases for this question illustrated that the prompt presented to the user should be shown just once, whereas the Codex response placed the printing of this prompt inside the loop that read input, thus printing it repeatedly.

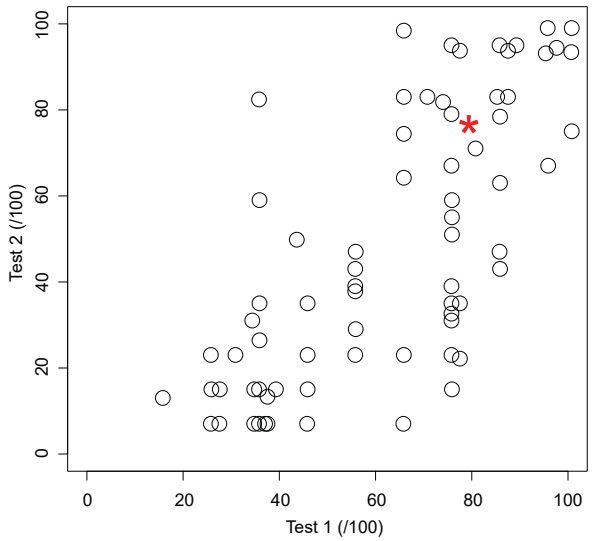

Figure 4: Student scores on invigilated tests (Test 1 and Test 2), with performance of Codex (plotted as red asterisk).

4.1.3 Failed Problems. In our evaluation, four problems remained unsolved after 10 consecutive attempts. The two problems from Test 1 (Q10, Q11) both placed restrictions on what features could be used in solutions (and the grading tool enforced these restrictions). For example, Q10 asked students to write a function that takes two

```
>>> print_triangle_numbers(5)
    1
   222
  33333
 4444444
555555555
```

```
>>> data_dict = {"A":5,"B":4,"C":2}
>>> draw_bar_graph(data_dict,5)
 ____
|    | ____
|    ||    |
|    ||    | ____
|    ||    ||    |
|    ||    ||    |
------------------
```

Figure 5: Examples shown as part of the problem descriptions, illustrating formatting requirements, for failed problems Q7 (left) and Q12 (right) from Test 2.

string parameters (a letter and a sentence) and "prints all the words in the sentence that start with the letter". However, solutions were required to "use a while loop to perform the task" and were not allowed to "use the split() method". While the solutions produced by Codex generally met both requirements – all ten solutions used a while loop, and only one attempted to use split() – they failed to produce the correct output. One common mistake was that for each matching word, the solution would print only the first letter of the word rather than the complete word. For Q11, students were asked to write a function that printed four integer parameters in sorted order only using the min() and max() functions and could not use conditionals, collections, or loops. Only two of the 10 solutions produced correct output, but both of these solutions violated this constraint.

Two problems from Test 2 (Q7, Q12) both placed quite specific formatting requirements on the output. Q7 asked students to define a function that would print an "isosceles triangle as in the examples below", given an integer input, without further description of the output. Q12 asked students to define a function that produced an ASCII histogram given a dictionary of values representing the bar heights. Figure 5 illustrates the required output for these two questions. The solutions generated by Codex for Q7 were very close to being correct – on several occasions producing the correct values but failing to print the leading spaces required on each row (we note that the problem description did not specifically state spaces were needed, but left the student to infer this from the examples). The solutions generated by Codex for Q12 tended to produce output that used the correct ASCII characters, but did not come close to the required format. Generating ASCII images is a challenging problem given that the output must be printed one row at a time.

4.2 Rainfall Problem Performance

Our second research question asks: "How does Codex perform on variations of a benchmark computing education problem that differ in context and level of detail?". The ten test cases used for the rainfall responses are described in Table 3 which shows examples of the standard input and list arguments used in the test cases. For the Guzdial [16] test cases, the sentinel value was not included in the list arguments as a sentinel was not mentioned in the question prompt. For the Lakanen et al. [18] test cases, the 'lowerLimit' was set to -1 and the 'sentinel' was set to 999.

The high-level results of the evaluation are shown in Table 4. Each response was graded against each test case and marked out of one (with one representing a perfect score — each test case contributes 0.1 towards the overall score). The average mark for each test case by variant is shown in Figure 6. For each test case, the binary pass/fail marks were averaged.

Name	Stdin	List Argument
Blank	S\	[S]
One 0	0\S\	[0, S]
One +ve	5\S\	[5, S]
One -ve	-5\S\	[-5, S]
Multiple +ve	3\5\7\S\	[3, 5, 7, S]
Multiple 0's	0\0\0\S\	[0, 0, 0, S]
Multiple -ve	-3\-5\-7\S\	[-3, -5, -7, S]
Mixed +ve & 0	4\0\S\	[4, 0, S]
Mixed +ve & -ve	3\-2\5\S\	[3, -2, 5, S]
Mixed All	3\0\-2\5\S\	[3, 0, -2, 5, S]

Table 3: Rainfall test cases (S→Sentinel, \→newline).

Variant	Mean	Median	Max	Stddev
Soloway [39]	0.63	0.90	1.00	0.40
Simon [37]	0.48	0.50	1.00	0.28
Fisler [10]	0.61	0.70	1.00	0.26
Ebrahimi [9]	0.19	0.05	1.00	0.26
Guzdial et al. [16]	0.47	0.30	1.00	0.22
Lakanen et al. [18]	0.44	0.70	0.90	0.32
apples	0.54	0.60	1.00	0.34

Table 4: Rainfall results marked out of 1 (max score).

When compared to the published partial scores (i.e. percentage of passing test cases – not all-or-nothing as described in section 3.1) from the literature, the performance of Codex is varied. Of the published variants, three reported partial scores. Codex outperformed the results of Simon [37] with 149 students using C# with an average partial score of 29% and none of the students providing a fully correct solution. Codex has similar results to Guzdial et al. [16] with 120 students using Python with an average partial score of 46%. Codex performs worse than the results of Lakanen et al. [18] where 139 students had an average partial mark of 69%. However, the test cases used to grade the published variants are likely more exhaustive than the tests used in our analysis, so it may not be an entirely fair comparison.

However, the Codex responses to the *apples* variant can be directly compared to the responses of our students. This variant was used in a CS1 course using Python with 45 students in 2019. The course is usually the second programming focused course computer science students take at our institution. The *apples* question was given to students in a midterm test under similar conditions to those reported in Section 3.1. Using the test cases provided in Table 3, the students received an average partial mark of 84%. However, in this setting students are able to re-attempt questions as described in 3.1.

Codex struggles on cases where no valid values are provided as input, with the Simon, Fisler, Ebrahimi, Guzdial, and Lakanen responses getting poor marks on the blank input and negative value test cases (i.e. cases where no non-negative values are provided). In these variants, the prompts excluded negative values but did not mention the case where no valid inputs are provided. However, the

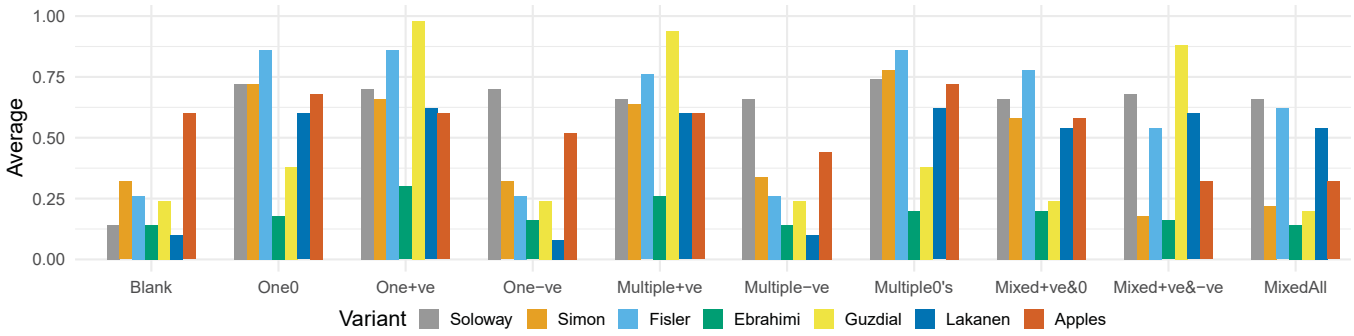

Figure 6: Mean mark (out of 1) per-case.

Variant	One while	One for	Sum / Len	Two Pass	Other
Soloway [39]	41	1	0	5	3
Simon [37]	5	36	1	8	0
Fisler [10]	3	42	1	3	1
Ebrahimi [9]	36	3	0	10	1
Guzdial et al. [16]	0	35	1	13	1
Lakanen et al. [18]	2	37	2	5	4
apples	4	23	3	16	4

Table 5: Count of general method used by response.

apples variant explicitly mentions to return -1 in the case there are no valid values and does much better on these cases in comparison.

4.3 Variety

Our third research question asks: "How much variety is there in the solutions generated by Codex?". To evaluate the variety of solutions generated by Codex, we examined the number of source lines of code (sloc) excluding blank and comment lines of solutions to all rainfall variants, as well as the general algorithmic approach employed in the solutions as an indicator of algorithmic variation. Figure 7 shows sloc. Table 5 reports on the algorithmic variation highlighting the different approaches of the solutions generated by Codex. *One while* and *one for* represent solutions that exclusively used a single while or for loop respectively. For example, using a single for loop to add values to a sum and increment a count on the condition a value is non-negative. *Sum/Len* represents solutions that calculated an average only using built-in sum() and len() functions without the use of loops. *Two pass* represents solutions that used multiple methods to clean and then aggregate the data. For example, having a while loop to filter values up to the sentinel followed by a for loop to compute the average. *Other* represents solutions that utilised some other method for calculating a result.

We find that the sloc and counts of general method indicate Codex is providing a range of different responses to the same prompt while ultimately favouring expected methods for each response (i.e., for loops for processing lists, and while loops for processing standard input). The variation is likely related to the high temperature value of 0.9 which the Codex documentation recommends as a value which will yield "creative" results.

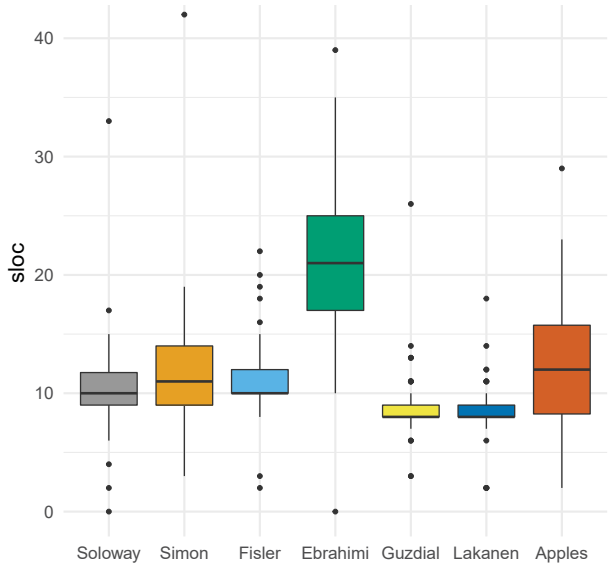

Figure 7: Source lines of code (sloc) per variant.

5 DISCUSSION

We cannot put the genie back in the bottle! AI is already capable of automatically generating a human-like quality of solutions in this context. We anticipate that in the near-term such tools will be able to generate solutions to increasingly more sophisticated problems, and will be more commonly used by students. Our results show that Codex performs better than most students on code writing questions in typical first year programming exams, and performs reasonably well in most variations of the Rainfall Problem. The solutions generated by Codex appear to include quite a lot of variation, which is likely to make it difficult for instructors to detect, but may offer some potential benefits for students. The question arising for the computing education community (perhaps the most significant question of the present century – so far at least) is how we engage with the challenges and opportunities presented by the increasing effectiveness of machine learning tools such as Codex.

5.1 Opportunities

Learning to program often involves a lot of relatively short programming exercises for students to become familiar with programming syntax, semantics, and style. Such activities are so common they may form a signature pedagogy [36] of computing education. Some of the exercises (or even previous years tests and exams) that are used for practice typically do not come with sample solutions, so automatically generated solutions may provide students with models that they can use for learning, or to check their own work.

The extensive literature on peer review, including the review of code [17, 21] makes it clear that there are many benefits that arise from looking at a variety of solutions to a given problem. These benefits are present even when the code is flawed (i.e., there are benefits from looking at poor solutions as well as good solutions). When presented as a way of generating solutions of *varying* quality, and alternative approaches that students are asked to evaluate, the automatically generated solutions from Codex may provide fertile ground for discussions of alternative approaches, quality of solutions, and potential for refactoring exercises.

5.2 Challenges

The availability of tools such as Codex raise concerns and challenges for the future of computing education. The value of a tool depends on its use, and there is the potential for Codex to be used in ways that limit learning, or ways that make the work of educators difficult. The developers of Codex mentioned one such challenge: possible over-reliance on Codex by novice programmers [5]. Here we discuss several other challenges in the context of introductory programming, and computing more generally.

Students may use Codex to generate model solutions for exercises where solutions are not provided by instructors. If the generated solution is incorrect or uses poor style, students are likely not learning optimally, and may adopt inappropriate conventions and poor style. While this is true for any crowd-sourced solution (i.e., using web-based examples), the customised solutions offered by Codex may lead students to perceive the solutions as being more credible, similar to an intelligent tutoring system [6].

This may suggest that future introductory programming courses should place more emphasis on code review, or evaluation of code, to ensure that students can effectively evaluate the quality of code generated by Codex rather than relying on such tools as an "oracle." This will mean an increasing reliance on invigilated exams in a time when academia is turning more and more toward online learning.

Finally, the fact Codex currently has limits in the complexity of problems it can solve may be irrelevant to students. If students submit code generated by Codex, even if it is not correct, it will likely be worthy of partial credit. How can an educator differentiate between a student who tried but ultimately failed to get the correct answer and a student who simply generated code using Codex? This may result in students being awarded a pass in a course without the appropriate level of knowledge required, and subsequently burden instructors (and peers if courses employ group projects).

5.3 Academic Integrity

Academics have long been concerned about violations of academic integrity in computing disciplines [8, 30]. The literature reports a high percentage of computing students to be engaged in some form of plagiarism [7]. Some studies have reported almost 80% of computing students to be involved [35], and there is a higher occurrence of plagiarism in computer science than in many other disciplines [20, 29, 30]. A recent systematic review of literature on plagiarism observes that there are attributes that are specific to computing, such as the nature of code reuse, that may impact on the academic dishonesty of computing students [1]. Interestingly, even though the study was published in 2019, it does not once mention the possibility of students utilising artificial intelligence to cheat.

For many years academics reported improvements in technology and availability of resources as being one of the most significant factors contributing to increased plagiarism [7]. Codex may further exacerbate this problem as well as blur the lines between generally accepted intelligent support provided some by IDEs, and activities that would be considered academic misconduct.

The use of many small exercises is an approach that CS1 students can prefer and have better success with [2]. Instructors may also be inclined to use them as they are easy to generate and assess automatically. However, this approach is likely to result in problems that can be more easily solved using Codex. This is contrasted with calls for the use of more complexity in assignments to reduce plagiarism [38]. It is possible that developments such as Codex may add to the need to change our common assessment practices in the direction of more unique and complicated assessments.

The availability of solutions to programming tasks can be problematic for student learning. Sheard et al. [34] make the case that: "If students are given tasks for which solutions are readily available to copy from textbooks or lecture notes they are tempted to take short cuts and avoid the intended learning experience." Solutions to common problems are easy to locate using standard web search engines. To avoid a solution being available, some educators may choose to introduce variations, or use descriptions that obfuscate the problem so an existing solution is more difficult to find. Evidence from our analysis shows that Codex was able to produce a solution to the *apples* problem with similar success to the more typical *Rainfall Problem*, despite the phrasing deliberately obscuring the similarity to readily available web-based solutions.

One method proposed to prevent copying is to have the instructor create unique assignments for each student [7, 24, 35]. Although this approach can prevent direct copying, students may still engage in "contract cheating" where work is outsourced to an individual or organisation [19, 25]. This approach frequently costs the student financially and has the potential for the solution to be shared with others using the same service, and subsequently become detected by similarity comparison tools. Codex generates new solutions each time, and the answers are not archived where they are visible to others, so it avoids the reproduction of identical discoverable solutions. This provides many of the benefits of contract cheating while avoiding the financial cost and the potential pitfalls of sharing a solution or potential identification by the contracted party.

Identifying plagiarism using similarity detection is a common solution [35], with a third of academics responding to a survey on academic integrity reporting the use of some form of similarity detection software [7]. Given the wide variation in student solutions generated by Codex, we anticipate that standard approaches to

```
"""
Write the get_rearranged_list() function which takes three parameters:
a list of numbers, an index value and an integer (how_many).  The
function returns a new list which has exactly the same elements as the
parameter list but with the elements rearranged so that the slice of
the list starting from the index parameter is moved to the end of the
new list. The length of the slice is given by the how_many parameter,
e.g. if the parameter list is [15, 10, 5, 20], the index value is 1 and
how_many is 2, then the two elements starting from index 1 (the
elements 10 and 5) are moved to the end of the new list.
"""

def get_rearranged_list(numbers:list, index:int, how_many:int) -> list:
    list_to_return = []

    for i in range(len(numbers)):
        if i not in range(index, index+how_many):
            list_to_return.append(numbers[i])

    for j in range(how_many):
        list_to_return.append(numbers[index+j])

    return  list_to_return

def get_rearranged_list(numbers, index , how_many):
    newlist = numbers[index:index + how_many]
    return numbers[:index] + numbers[index + how_many:] + newlist
```

Figure 8: Problem description for Test 2 Q6 (top, green font between """) and two correct solutions produced by Codex (each beginning with 'def').

identifying copying through similarity detection will be largely unsuccessful, and a different approach may be needed.

Figure 8 provides an example of the variety we observed in solutions to problems. The problem description is from Test 2 Q6, beneath which are two distinct solutions produced by Codex, both of which correctly solve the problem. Note the inclusion of type hints in the first solution, and the use of list slicing in the second.

Codex provides students with a means of obtaining solutions to simple programming problems that carries low risk of similarity detection due to the variation in solutions, and no risk of the activity being revealed by a human who has provided the solution. Given the difficulty involved in the detection of Codex solutions, we suggest that a strategy focused on education is worthwhile. Albluwi [1] states that there is confusion among students and no consensus among instructors on what constitutes plagiarism in programming assessments. The use of Codex and similar tools may further complicate this issue. We suggest that there may be benefits from explicitly discussing the use of Codex and other similar tools in class, alongside other academic integrity guidelines.

5.4 Changes Ahead

Technologies such as Codex will improve and proliferate. These tools will change the way we teach, learn, and work. Floridi and Chiriatti [11] predict that "People whose jobs still consist in writing will be supported, increasingly, by tools such as GPT-3. Forget the mere cut & paste, they will need to be good at prompt & collate." Much as consumers of news articles and other text will have to get used to not knowing what they are reading was authored by humans or generated by AI tools [11], programmers will soon (if not already) not know if code they have not seen before was written by a human, an AI tool, or a combination of both. Trying to gauge the intention or reasoning of a program's author from their code is difficult. It is arguably pointless if the author was not human.

We are also likely to see technologies such as Codex used as a component in other tools, aiding the proliferation of low-code / no-code platforms and their use. The emergence of Codex raises several questions for the computing education community, and many avenues to explore in future work:

- What problem types are difficult for tools such as GPT-3 and Codex? What is the performance of such tools on Parsons problems, MCQs, and problems with contextual specifications.
- How does Codex perform on other question types such as "Explain in plain English", identifying bugs in code, and fact-based questions (e.g., list all the identifiers in the code provided)?
- Can automated plagiarism detection tools identify code generated by Codex?
- Can tools like Codex be utilised to detect plagiarism?
- How can Codex be used to improve student learning?
- How should we adapt course content and assessment approaches as the use of tools such as Codex becomes more prevalent?

We believe that further work investigating the challenges and opportunities presented by Codex and similar tools is of urgent importance for the computing education community.

6 LIMITATIONS

The data that Codex was trained on may have included solutions to the previously published Rainfall Problem. However this would be at least partially mitigated by the fact that we used our own unique variant of Rainfall, and the use of test questions which weren't published. Additionally it is possible that some of the solutions from our test questions were posted online by students and included in the Codex training data. This may be mitigated in part by the fact that we used a high temperature value (0.9 out of a maximum of 1) in Codex which produces more random (i.e. "creative") responses, and we did observe a good degree of variety in the Codex responses.

7 CONCLUSIONS

In this paper, we have examined what could be considered an emergent existential threat to the teaching and learning of introductory programming. With some hesitation we have decided to state how we truly feel – the results are stunning – demonstrating capability far beyond that expected by the authors.

Having Codex in the hands of students should warrant concern similar to having a power tool in the hands of an amateur. The tool itself may not be intended to do harm, but with a vulnerable or untrained user, it may do just that. Even though, as we have shown above, the current tool has clear limitations when given tight constraints, it is evident from the rapid progression of GPT-2 to GPT-3 to Codex, that this will not remain the case for long. Furthermore, total correctness need not be a reality for technologies such as this to shake the foundations of computing education. The way we teach introductory programming – and probably eventually all of computing – will change drastically in the next decade, and the largest driver of that change may be tools such as Codex.

However, all is not lost. While tools like Codex present clear threats and challenges to student learning and academic integrity, they also present fantastic opportunities to refactor existing curricula. The CS1 of the future might use tools like Codex to create unique code snippets for each student to analyse on invigilated

exams; students could be provided with a window into the variety of solutions for any given problem; and eventually tools like Codex may assist in automatic evaluation of student code.

Whatever we do, it is certain that this particular AI revolution has arrived at the door of our classrooms, and we must consider how we adapt to it. Keeping it out is not an option. We expect much future work as tools like Codex appear in classrooms globally. Anticipating this shift, we put the following sentence into GPT-3, "The robots are taking over." It returned, "Yes, you read that right. The robots are coming for us. We have been warned".

REFERENCES

[1] Ibrahim Albluwi. 2019. Plagiarism in Programming Assessments: A Systematic Review. *ACM Trans. Comput. Educ.* 20, 1, Article 6 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371156

[2] Joe Michael Allen, Frank Vahid, Alex Edgcomb, Kelly Downey, and Kris Miller. 2019. An Analysis of Using Many Small Programs in CS1. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, NY, NY, USA, 585–591. https://doi.org/10.1145/3287324.3287466

[3] Brett A. Becker and Keith Quille. 2019. 50 Years of CS1 at SIGCSE: A Review of the Evolution of Introductory Programming Education Research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, NY, NY, USA, 338–344. https://doi.org/10.1145/3287324.3287432

[4] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, et al. 2020. Language Models Are Few-shot Learners. *arXiv preprint arXiv:2005.14165* (2020).

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:cs.LG/2107.03374 https://arxiv.org/abs/2107.03374

[6] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent Tutoring Systems for Programming Education: A Systematic Review. In *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*. ACM, NY, NY, USA, 53–62. https://doi.org/10.1145/3160489.3160492

[7] Martin Dick, Judy Sheard, Cathy Bareiss, Janet Carter, Donald Joyce, et al. 2002. Addressing Student Cheating: Definitions and Solutions. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '02)*. ACM, NY, NY, USA, 172–184. https://doi.org/10.1145/960568.783000

[8] John L. Donaldson, Ann-Marie Lancaster, and Paula H. Sposato. 1981. A Plagiarism Detection System. *SIGCSE Bull.* 13, 1 (Feb. 1981), 21–25. https://doi.org/10.1145/953049.800955

[9] Alireza Ebrahimi. 1994. Novice Programmer Errors: Language Constructs and Plan Composition. *Int. J. Hum.-Comput. Stud.* 41, 4 (Oct. 1994), 457–480. https://doi.org/10.1006/ijhc.1994.1069

[10] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14)*. ACM, NY, NY, USA, 35–42. https://doi.org/10.1145/2632320.2632346

[11] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its Nature, Scope, Limits, and Consequences. *Minds and Machines* 30, 4 (2020), 681–694. https://doi.org/10.1007/s11023-020-09548-1

[12] Lex Fridman. 2021. Donald Knuth: Programming, Algorithms, Hard Problems & the Game of Life | Lex Fridman Podcast #219. https://www.youtube.com/watch?v=EE1R8FYUJm0&t=1995s

[13] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. 2020. The State of the ML-Universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. ACM, NY, NY, USA, 431–442. https://doi.org/10.1145/3379597.3387473

[14] Mark Guzdial. 2011. From Science to Engineering. *Commun. ACM* 54, 2 (Feb. 2011), 37–39. https://doi.org/10.1145/1897816.1897831

[15] Mark Guzdial. 2013. Exploring Hypotheses about Media Computation. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)*. ACM, NY, NY, USA, 19–26. https://doi.org/10.1145/2493394.2493397

[16] Mark Guzdial, Rachel Fithian, Andrea Forte, and Lauren Rich. 2003. Report on Pilot Offering of CS1315 Introduction to Media Computation With Comparison to CS1321 and COE1361.

[17] Theresia Devi Indriasari, Andrew Luxton-Reilly, and Paul Denny. 2020. A Review of Peer Code Review in Higher Education. *ACM Trans. Comput. Educ.* 20, 3, Article 22 (Sept. 2020), 25 pages. https://doi.org/10.1145/3403935

[18] Antti-Jussi Lakanen, Vesa Lappalainen, and Ville Isomöttönen. 2015. Revisiting Rainfall to Explore Exam Questions and Performance on CS1. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15)*. ACM, NY, NY, USA, 40–49. https://doi.org/10.1145/2828959.2828970

[19] Thomas Lancaster and Codrin Cotarlan. 2021. Contract Cheating by STEM Students Through a File Sharing Website: A Covid-19 Pandemic Perspective. *International Journal for Educational Integrity* 17, 1 (2021), 1–16.

[20] Alberta Lipson and Norma McGavern. 1993. Undergraduate Academic Dishonesty at MIT. Results of a Study of Attitudes and Behavior of Undergraduates, Faculty, and Graduate Teaching Assistants. (1993).

[21] Andrew Luxton-Reilly. 2009. A Systematic Review of Tools That Support Peer Assessment. *Computer Science Education* 19, 4 (2009), 209–232.

[22] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, et al. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018 Companion)*. ACM, NY, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[23] Zohar Manna and Richard J. Waldinger. 1971. Toward Automatic Program Synthesis. *Commun. ACM* 14, 3 (March 1971), 151–165. https://doi.org/10.1145/362566.362568

[24] Sathiamoorthy Manoharan. 2017. Personalized Assessment as a Means to Mitigate Plagiarism. *IEEE Transactions on Education* 60, 2 (2017), 112–119. https://doi.org/10.1109/TE.2016.2604210

[25] Sathiamoorthy Manoharan and Ulrich Speidel. 2020. Contract Cheating in Computer Science: A Case Study. In *2020 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. 91–98. https://doi.org/10.1109/TALE48869.2020.9368454

[26] Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. 2002. The Effects of Pair-Programming on Performance in an Introductory Programming Course. *SIGCSE Bull.* 34, 1 (Feb. 2002), 38–42. https://doi.org/10.1145/563517.563353

[27] Cade Metz. 2021. A.I. Can Now Write Its Own Computer Code. Thats Good News for Humans. https://www.nytimes.com/2021/09/09/technology/codex-artificial-intelligence-coding.html

[28] OpenAI. 2020. About OpenAI. https://openai.com/about/

[29] Paul Phillips and Luc Cohen. 2014. Convictions of Plagiarism in Computer Science Courses on the Rise. *The Daily Princetonian, March* 4 (2014), 2014.

[30] Eric Roberts. 2002. Strategies for Promoting Academic Integrity in CS Courses. In *32nd Annual Frontiers in Education*, Vol. 2. IEEE, F3G–F3G. https://doi.org/10.1109/FIE.2002.1158209

[31] Kevin Scott. 2020. Microsoft teams up with OpenAI to Exclusively License GPT-3 Language Model. https://blogs.microsoft.com/blog/2020/09/22/microsoft-teams-up-with-openai-to-exclusively-license-gpt-3-language-model/

[32] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do We Know How Difficult the Rainfall Problem Is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15)*. ACM, NY, NY, USA, 87–96. https://doi.org/10.1145/2828959.2828963

[33] Sam Shead. 2021. Why Everyone is Talking About an Image Generator Released by an Elon Musk-Backed A.I. Lab. https://www.cnbc.com/2021/01/08/openai-shows-off-dall-e-image-generator-after-gpt-3.html

[34] Judy Sheard, Angela Carbone, and Martin Dick. 2003. Determination of Factors Which Impact on IT Students' Propensity to Cheat. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20 (ACE '03)*. Australian Computer Society, Inc., AUS, 119–126.

[35] Judy Sheard, Simon, Matthew Butler, Katrina Falkner, Michael Morgan, et al. 2017. Strategies for Maintaining Academic Integrity in First-Year Computing Courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, NY, NY, USA, 244–249. https://doi.org/10.1145/3059009.3059064

[36] Lee S. Shulman. 2005. Signature Pedagogies in the Professions. *Daedalus* 134, 3 (2005), 52–59. http://www.jstor.org/stable/20027998

[37] Simon. 2013. Soloway's Rainfall Problem Has Become Harder. In *2013 Learning and Teaching in Computing and Engineering*. 130–135. https://doi.org/10.1109/LaTiCE.2013.44

[38] Simon. 2017. Designing Programming Assignments to Reduce the Likelihood of Cheating. In *Proceedings of the 19th Australasian Computing Education Conference (ACE '17)*. ACM, NY, NY, USA, 42–47. https://doi.org/10.1145/3013499.3013507

[39] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858. https://doi.org/10.1145/6592.6594

[40] Alex Tamkin, Miles Brundage, Jack Clark, and Deep Ganguli. 2021. Understanding the Capabilities, Limitations, and Societal Impact of Large Language Models. *arXiv preprint arXiv:2102.02503* (2021).

[41] Laurie A. Williams and Robert R. Kessler. 2000. All I Really Need to Know about Pair Programming I Learned in Kindergarten. *Commun. ACM* 43, 5 (May 2000), 108–114. https://doi.org/10.1145/332833.332848

[42] Wojciech Zaremba, Greg Brockman, and OpenAI. 2021. OpenAI Codex. https://openai.com/blog/openai-codex/