Computing Education in the Era of Generative AI

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

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

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

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

Arto Hellas
Aalto University
Espoo, Finland
arto.hellas@aalto.fi

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

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

Brent N. Reeves
brent.reeves@acu.edu
Abilene Christian University
Abilene, Texas, USA

Eddie Antonio Santos
University College Dublin
Dublin, Ireland
eddie.santos@ucdconnect.ie

Sami Sarsa
Aalto University
Espoo, Finland
sami.sarsa@aalto.fi

ABSTRACT

The computing education community has a rich history of pedagogical innovation with many efforts, especially at the introductory level, focused on helping students learn how to program. Recent advances in artificial intelligence have led to large language models that can produce source code from natural language problem descriptions with impressive accuracy. The wide availability of these models and their ease of use is raising urgent questions for educators around the need to adapt their pedagogy as well as prompting broader discussion on the computing curriculum. In this article, we discuss the challenges and the opportunities that such models present to computing educators, with a focus on introductory programming classrooms. We organize this discussion around two foundational articles from the computing education literature that were written around the time that awareness of code generating language models was just beginning to emerge. The first of these (in January 2022) evaluated the performance of code generating models on typical introductory-level programming problems, and the second (in August 2022) explored the quality and novelty of learning resources generated by these models. Now, less than two years since the first article was published, we consider implications for computing education in light of new model capabilities and as lessons emerge from educators incorporating such models into their teaching practice.

CACM, Jan, 2023, Magazine
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-xxxx-xxxx-x/xx/xx.
https://doi.org/10.1145/xxxxxx.xxxxxx

CCS CONCEPTS

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

KEYWORDS

academic integrity; AI; artificial intelligence; code generation; code writing; Codex; computer programming; Copilot; CS1; deep learning; generatve AI; introductory programming; GitHub; GPT-3; large language models; machine learning; ML; neural networks; natural language processing; novice programming; OpenAI

ACM Reference Format:
Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2022. Computing Education in the Era of Generative AI. In *Communications of the ACM (CACM), Magazine, 2023* . ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/xxxxxx.xxxxxx

1 INTRODUCTION

A new era is emerging in which artificial intelligence will play an ever-increasing role in many facets of daily life. One defining characteristic of this new era is the ease with which novel content can be generated. Large language models (LLMs), neural network-based models trained on vast quantities of text data [4], are capable of creating a variety of convincing human-like outputs, including prose, poetry, and source code. It is largely accepted that synthesizing source code automatically from natural language prompts is likely to improve the productivity of professional developers [26], and is being actively explored by well-funded entities such as OpenAI (ChatGPT, GPT-4[1]), Amazon (CodeWhisperer[2]), and Google (AlphaCode [21], Bard[3]). In the same way that high-level programming

[1]openai.com/research/gpt-4
[2]aws.amazon.com/codewhisperer
[3]blog.google/technology/ai/code-with-bard

languages offered large productivity advantages over assembly language programming in the 1970s, AI code generation tools look set to transform traditional programming practices. Already, claims are emerging that a significant proportion of new code is being produced by tools such as GitHub Copilot [9], a plug-in for popular IDEs such as Visual Studio Code.

The current pace of development in this area is staggering with noticeably more advanced model versions being released several times per year. The pace of advancement is so rapid that in March 2023, a well-publicized open letter appeared that encouraged a public, verifiable, and immediate pause of at least six months duration on the training of AI systems more powerful than GPT-4. Signed by Elon Musk, Steve Wozniak, Moshe Vardi and thousands of others including many AI-leaders and Turing award winners[4], the letter was addressed to all AI labs and suggested potential government-led moratoriums.

These developments raise urgent questions about the future direction of many aspects of society, including computing education. For example, one popular evidence-based pedagogy for teaching introductory programming involves students writing many small exercises that are checked either manually or by automated grading tools. However, these small problems can now easily be solved by AI models. Often, all that is required of a student is to accept an auto-generated suggestion by an IDE plugin [10, 11]. This raises concerns that students may use new tools in ways that limit learning and make the work of educators more difficult. Bommasani et al. highlight that it will become "much more complex for teachers to understand the extent of a student's contribution" and to "regulate ineffective collaborations and detect plagiarism" [4]. Alongside such challenges come emerging opportunities for students to learn computing skills [2].

In this article, we consider the implications of generative AI on computing education, and explore how newly emerging tools are likely to impact students and educators in introductory programming classrooms. We organize this article into two main sections: challenges and opportunities. With respect to challenges, we illustrate the performance of code generation models on typical introductory-level programming problems and discuss issues relating to plagiarism, learner over-reliance, and potential risks around bias and bad habits. With respect to opportunities, we illustrate how these models can be used to generate learning resources, including programming exercises and code explanations, and further discuss the potential for improving feedback to students, such as error message reporting, and new pedagogical approaches.

1.1 Large Language Models and Code

AI-driven coding has only been a viable reality for the general public since 2022, when GitHub's Copilot emerged from a period of technical preview. Originally pitched as "your AI pair programmer", at the time of writing, Copilot claims to be the "world's most widely adopted AI developer tool"[5]. Other AI-powered code generation tools are also broadly accessible, including Amazon's CodeWhisperer and Google's Bard. The Codex model (discussed in this article specifically) was the original model to power Copilot. A descendant of GPT-3, Codex was fine-tuned with code from over 50 million public GitHub repositories totaling 159 GB [5]. Although now officially deprecated in favor of the newer chat models, Codex was capable of taking English-language prompts and generating code in several programming and scripting languages, including JavaScript, Go, Perl, PHP, Python Ruby, Swift, TypeScript, shell and more.

It could also translate code between programming languages, explain (in several natural languages) the functionality of code, and return the time complexity of the code it generated.

The use of such tools in education is nascent and changing rapidly. Copilot was only made freely available to students in June 2022[6], and to teachers in September 2022 after its potential to impact education began to unfold[7]. In November 2022, ChatGPT[8] was released, followed by the release of GPT-4 in March 2023. OpenAI has continued to update these models with new features, such as data analysis from files, analyzing images, and assisted web search. For a more technical overview of the historical developments and future trends of language models, we direct the reader to the CACM article by Li [20].

2 CHALLENGES AHEAD

Code generation tools powered by LLMs appear to be able to correctly and reliably solve many programming problems that are typical in introductory courses. This raises a number of important questions for educators. For example, just how good are these tools? Can a student with no programming knowledge, but who is armed with a code-generating LLM, pass typical programming assessments? Do we need a different approach?

2.1 Putting Them to the Test

To explore the performance of LLMs in the context of introductory programming, we prompted Codex with real exam questions and compared its performance to that of students taking the same exams. We also prompted Codex to solve several variants of a well-known CS1-level programming problem (the "Rainfall problem") and examined both the correctness and the variety of solutions produced. This work was originally performed in September 2021, several weeks after OpenAI provided API access to the Codex model. The resulting paper, published in January 2022, was the first in a computing education venue to document the code-generating capabilities of LLMs [10].

2.1.1 My AI wants to know if its grade will be rounded up. We took all questions from two Python CS1 programming exams that had already been taken by students and provided them as input (verbatim) to Codex. The exam questions involved common Python datatypes including strings, tuples, lists, and dictionaries. They ranged in complexity from simple calculations, such as computing the sum of a series of simulated dice rolls, to more complex data manipulations such as extracting a sorted list of the keys that are mapped to the maximum value in a dictionary.

To evaluate the code generated, we executed it against the same set of test cases that were used in assessing the student exams. This follows a similar evaluation approach employed by the Codex

[4]futureoflife.org/open-letter/pause-giant-ai-experiments
[5]github.com/features/copilot
[6]github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers
[7]github.blog/2022-09-08-github-copilot-now-available-for-teachers
[8]openai.com/blog/chatgpt

developers [5]. If the Codex output differed from the expected output with only a trivial formatting error (for example, a missing comma or period) we made the appropriate correction, much as a student would if using Codex to complete an exam.

To contextualize the performance of the Codex model, we calculated the score for its responses in the same way as for real students using the same question weights and accumulated penalties for incorrect submissions. Codex scored 15.7/20 (78.5%) on Exam 1 and 19.5/25 (78.0%) on Exam 2. Figure 1 plots the scores (scaled to a maximum of 100) of 71 students enrolled in the CS1 course in 2020 who completed both exams. Codex's score is marked with a blue 'X'. Averaging both Exam 1 and Exam 2 performance, Codex ranks 17 amongst the 71 students, placing it within the top quartile of class performance.

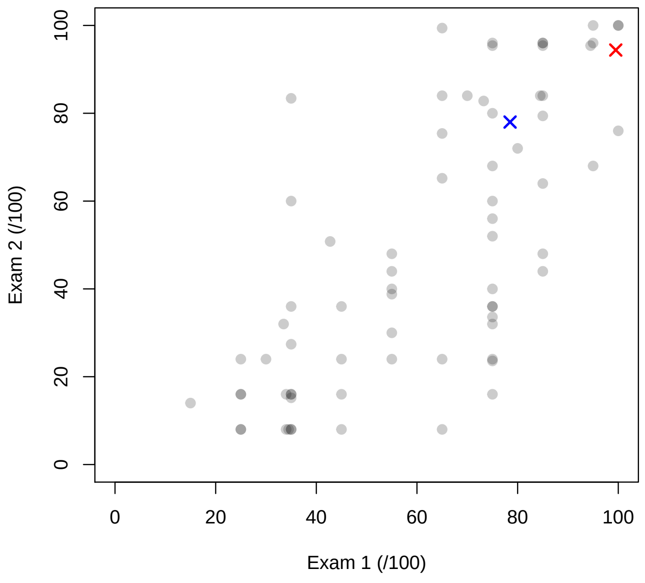


Figure 1: Student scores on Exam 1 and Exam 2, represented by circles. Codex's 2021 score is represented by the blue 'X'. GPT-4's 2023 score on the same questions is represented by the red 'X'.

We observed that some of the Codex answers contained trivial formatting errors. We also observed that Codex performed poorly with problems that disallowed the use of certain language features (e.g. using `split()` to tokenize a string). Codex often did not produce code that avoided using these restricted features, and thus the model (in these cases) often did not pass the auto-grader. Codex also performed poorly when asked to produce formatted ASCII output such as patterns of characters forming geometric shapes, especially where the requirements were not specified in the problem description but had to be inferred from the provided example inputs and outputs.

2.1.2 Yes, I definitely wrote this code myself. To understand the amount of variation in the responses, we provided Codex with seven variants of the problem description for the well-studied 'Rainfall' problem (which averages values in a collection) a total of 50 times each, generating 350 responses. Each response was executed against 10 test cases (a total of 3500 evaluations). Across all variants, Codex had an average score close to 50%. Codex performed poorly on cases where no valid values were provided as input (e.g. where the collection to be averaged was empty).

We also examined the number of source lines of code for all Rainfall variants, excluding blank and comment lines. In addition, we classified the general algorithmic approach employed in the solutions as an indicator of algorithmic variation. We found that Codex provides a diverse range of responses to the same input prompt. Depending on the prompt, the resulting programs used varied programmatic structures, while ultimately favoring expected methods for each problem variation (i.e., for-loops for processing lists, and while-loops for processing standard input).

2.1.3 Rapid progress. Given the improvement in model capabilities over the last two years, it is interesting to observe how well a state-of-the-art model (GPT-4 at the time of writing) performs on the same set of questions. In July 2023, a working group exploring LLMs in the context of computing education replicated this study using GPT-4 under identical conditions [29]. GPT-4 scored 99.5% on Exam 1 and 94.4% on Exam 2, this time outscored overall by only three of the 71 students (represented by the red 'X' in Figure 1). On the Rainfall problems, GPT-4 successfully solved every variant, in some cases producing the correct result but with a trivial formatting error. Another follow-up study looked at the performance of generative AI on CS2 exam questions and found that it performed quite well in that context [11]. Newer models can also solve other types of programming exercises, like Parsons Problems, with decent accuracy that is likely to only increase over time [31].

2.2 Academic Integrity

Software development often encourages code reuse and collaborative development practices, which makes the concept of academic integrity difficult to formalize in computing [35]. Nevertheless, individual work is still commonplace in computing courses, and it is an expectation for students working on individual projects to produce their own code rather than copying code written by someone else. This is often verified through the use of traditional plagiarism tools. However, recent work has shown that common plagiarism detection tools are often ineffective against AI-generated solutions [3]. This raises significant concerns for educators monitoring academic integrity in formal assessments.

2.2.1 Academic misconduct. Although academic misconduct has been discussed in the computing education community for quite some time [35], the advent of LLMs provides a new and difficult set of challenges. The first is categorizing exactly what type of academic misconduct, if any, its usage falls into. A recent working group report on LLMs in computing education considered ethics and examined it in the context of the ACM Code of Ethics and recent university AI usage policies [29]. They discussed plagiarism, collusion, contract cheating, falsification, and the use of unauthorized resources. Although many university policies have placed AI usage into the category of plagiarism, Prather et al. disagree [29]. Plagiarism steals content from a person with agency, which AI resources as next-token-generators do not have. If generative AI tools are seen as productivity tools (such as IDE code-completion or calculators for mathematical problems) that are used professionally, then it makes sense to decide if the use of such tools is appropriate for a given context and communicate the decision to students. If students persist in using the tools when they are restricted, then they would be engaging in academic misconduct because they used an unauthorized resource and not because of some intrinsic characteristic of the tool itself. Instructors should, therefore, be extremely clear about when and how generative AI tools are allowed to be used on their assessments. The working group report includes a guide for students (Appendix D) that could easily be adapted by faculty into a helpful handout or added to a course syllabus.

A recent interview study with computing educators has revealed that initial reactions are divided – from banning all use of generative AI to an acceptance that resistance is, ultimately, futile [17]. Restricting the use of generative AI tools is likely (at least in the short term) to shift practice towards increased use of secure testing environments [43], and a greater focus on the development and assessment of process skills [16].

2.2.2 Code reuse and licensing. Potential licensing issues arise when content is produced using code generation models, even when the model data is publicly-available [21]. Many different licenses apply to much of the publicly-available code used to train LLMs, and typically these licenses require authors to credit the code they used, even when the code is open-source. When a developer generates code using an AI model, they may end up using code that requires license compliance without being aware of it. Such issues are already being worked out in court[9]. This is clearly an issue that extends beyond educational use of software, but as educators it is our role to inform students of their professional responsibilities when reusing code.

2.3 Learner Over-reliance

The developers of Codex noted that a key risk of using code generation models in practice is users' over-reliance [5]. Novices using such models, especially with tools such as Copilot that embed support in an IDE, may quickly become accustomed to auto-suggested solutions. This could have multiple negative effects on student learning.

2.3.1 Metacognition. Developing computational thinking skills is important for novice programmers as it can foster higher-order thinking and reflection [23]. Metacognition, or "thinking about thinking", is a key aspect of computational thinking (and problem-solving in general) and has been shown to be closely related to it. While learning to code is already a challenging process that requires a high level of cognitive effort to remember language syntax, think computationally, and understand domain-specific knowledge, the use of metacognitive knowledge and strategies can aid in problem-solving and prevent beginners from getting overwhelmed or lost. Relying too heavily on code generation tools may hinder the development of these crucial metacognitive skills.

2.3.2 When the models fail. An analysis of solutions generated by AlphaCode revealed that 11% of Python solutions were syntactically incorrect (produced a `SyntaxError`) and 35% of C++ solutions did not compile [21]. Recent work has shown that as many as 20% of introductory programming problems are not solved sufficiently by code generation models, even when allowing for expert modification of the natural language problem descriptions [6]. The developers of Codex noted that it can recommend syntactically incorrect code, including variables, functions, and attributes that are undefined or outside the scope of the codebase, stating "Codex may suggest solutions that superficially appear correct but do not actually perform the task the user intended. This could particularly affect novice programmers and could have significant safety implications depending on the context" [5]. Students who have become overly reliant on model outputs may find it especially challenging to proceed when the suggested code is incorrect and cannot be resolved through natural language prompting [15].

2.4 Bias and Bad Habits

The issue of bias in AI is well known. In addition to general bias (subtle or overt) that applies to almost all AI-generated outputs, such as the representation of certain groups of people, genders, etc., there are likely biases specific to AI code generation.

2.4.1 Appropriateness for beginners. Novices usually start by learning simple programming concepts and patterns, gradually building their skills. However, much of the vast quantity of code on which these AI models are trained was written by experienced developers. Therefore, we should expect that AI generated code may sometimes be too advanced or complex for novices to understand and modify. Recent work has shown that even the latest generative AI models continue to generate code utilizing concepts too advanced for novices or that are specifically outside the curriculum [15].

2.4.2 Harmful biases. The developers of Codex found that code generation models raise bias and representation issues — notably that Codex can generate code comments (and potentially identifier names) that reflect negative stereotypes about gender and race, and may include other denigratory outputs [5]. Such biases are obviously problematic, especially where novices are relying on the outputs for learning purposes. Notably, the feature list for Amazon CodeWhisperer includes capabilities to remove harmful biases from generated code[10]. Some recent work (from competitor Microsoft) has expressed doubt about the reliability of this feature [33].

2.4.3 Security. Unsurprisingly, AI-generated code can be insecure [25], and human oversight is required for the safe use of AI code generation systems [5]. However, novice programmers lack the knowledge to provide this oversight. Perry et al. recently examined whether novices using AI code generation tools wrote more secure code, finding that novices consistently wrote insecure code with specific vulnerabilities in string encryption and SQL injection [27]. Perhaps even more disturbing, novice programmers in their study who had access to an AI code generating tool were more likely to believe they had written secure code. This reveals a pressing need for increased student and educator awareness around the limitations of current models for generating secure code.

[9]githubcopilotlitigation.com

[10]aws.amazon.com/codewhisperer/features

2.5 Computers in Society

The use of AI-generated code provides many opportunities for discussions on ethics and the use of computers in society. Moreover, these technologies may serve as a vehicle to empower novice users to explore more advanced ideas earlier, leveraging the natural engagement that comes from utilizing technologies that are "in the news". Teachers of introductory courses have long told themselves that students will learn about testing, security, and other more advanced topics in subsequent courses. However, with growing numbers of students taking introductory classes but not majoring in computing, and the capabilities that code generation affords, the stakes are higher for CS1 and introductory classes to raise these issues early, before the chance of real-world harm is great.

3 OPPORTUNITIES AHEAD

Despite the challenges that must be navigated, code generation tools have the potential to revolutionize teaching and learning in the field of computing [2]. Indeed, the developers of such models specifically highlight their potential to positively impact education. When introducing Codex, Chen et al. outline a range of possible benefits, including to: "aid in education and exploration" [5]. Similarly, the developers of AlphaCode suggest such tools have "the potential for a positive, transformative impact on society, with a wide range of applications including computer science education" [21]. In this section we discuss several concrete opportunities for code and text generation models to have a transformative effect on computing education.

3.1 Plentiful Learning Resources

Introductory programming courses typically utilize a wide variety of learning resources. For example, programming exercises are a very common type of resource for helping students practice writing code. Similarly, natural language explanations of code are another useful resource. They can be valuable for helping students understand how a complex piece of code works, or as a tool for evaluating student comprehension of code. However, it is a significant challenge for educators to generate a wide variety of high-quality exercises that are targeted to the interests of individual learners, and to produce detailed explanations at different levels of abstraction for numerous code examples.

We explored the potential for LLMs to reduce the effort needed by instructors to generate the two types of learning resources just discussed: programming exercises and code explanations. This work, which was originally carried out in April 2022 and published in August 2022, was the first paper in a computing education venue to explore LLM-generated learning resources [34].

3.1.1 Programming exercises. Figure 2 shows an example of the input we used to generate new programming exercises using Codex. This 'priming' exercise consists of a one-shot example (a complete example similar to the desired output) followed by a partial prompt to prime the generation of a new output. In this case, the format of the priming exercise consists of a label (Exercise 1) followed by keywords for both the contextual themes (i.e. donuts) and the programming-related concepts (i.e. function, conditional) of the exercise, a natural language problem statement and a solution (in the form of a Python function). For space reasons, we omit a list of test cases but these can also be included for programming problems. The priming input ends with the explicit prompt for a new exercise to be generated (Exercise 2), along with the desired concepts and themes expressed as keywords (i.e. basketball, function, list, and for loop).

Figure 3 shows one output generated when the prompt in Figure 2 was provided to Codex. Note that in this case, as requested by the keyword information in the input prompt, the problem statement is related to basketball and the model solution consists of a *function* that involves a *list* and a *for* loop. To evaluate this approach more thoroughly, we generated a set of 240 programming exercises by varying the programming related concepts and contextual themes. We attempted to execute the generated code against the generated test cases, and analyzed statement coverage as a measure of the thoroughness of the test suite. Table 1 summarizes these results, and shows that in most cases the programming exercises generated by the model included sample solutions that were executable. Similarly, most of the time the model also generated a set of tests, resulting in a total of 165 programming exercises with both a sample solution and a set of tests. The sample solution frequently did not pass all of the generated tests, but in cases when it did, the test suites achieved full statement coverage in all but three cases.

We also found that the vast majority of exercises (around 80%) were entirely novel in that fragments of the problem descriptions were not indexed by any search engines. A similar fraction of the exercises also matched the desired topics and themes. Although this is far from perfect, there is obvious potential for generating

```
"""Exercise 1
--Keywords--
donuts
function
conditional
--Problem statement--
Write a function called donut_cost that takes
 ↪  three inputs: the price of a donut, the
 ↪  number of donuts bought, and whether or not
 ↪  it is the weekend.  The function should
 ↪  return the total cost of the donuts.  If it
 ↪  is the weekend, the donuts cost twice as
 ↪  much.
--Sample solution--
def donut_cost(price, number, is_weekend):
  if is_weekend:
    return price * number * 2
  else:
    return price * number
"""Exercise 2
--Keywords--
basketball
function
list
for loop
--Problem statement--
```

Figure 2: A 'priming' exercise consisting of one complete example followed by the prompt for a new one.

```
Write a function called count_rebounds that
 ↪  takes a list of basketball players as an
 ↪  input.  The function should return the
 ↪  total number of rebounds for the entire
 ↪  team.  Each element in the list is itself a
 ↪  list containing the player's name, their
 ↪  points, and their rebounds.
--Sample solution--
def count_rebounds(players):
  total = 0
  for player in players:
    total = total + player[2]
  return total
```

Figure 3: Example output generated by Codex using the priming exercise from Figure 2.

Table 1: Analysis of 240 generated programming exercises, showing the proportion that met criteria regarding sample solutions and test cases.

Has sample solution?	203/240	84.6%
Sample solution executable?	182/203	89.7%
Has test cases?	170/240	70.8%
All tests pass?	51/165	30.9%
Full (100%) statement coverage?	48/51	94.1%

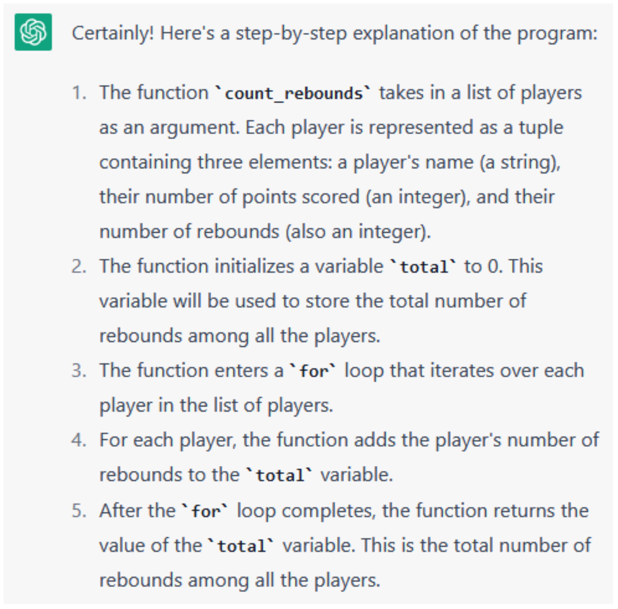
new and useful resources in this manner and the cost of eliminating poor results (which could be automated) is almost certainly smaller than generating a large number of exercises manually. With the addition of filtering steps that could be automated, it would be possible to generate an almost endless supply of novel resources that are contextualized to students' interests.

3.1.2 Code explanations. Code explanations can be generated at different levels of abstraction, from high-level summaries to detailed explanations of every line. We focused on the latter as these are often useful for students when debugging code. We prompted Codex using a simple input that consisted of the source code to be explained, the text "Step-by-step explanation of the above program", and finally "1." to influence the output to include numbered lines. We analyzed the resulting explanations in terms of completeness and accuracy, finding that 90% of the explanations covered all parts of the code, and nearly 70% of the explanations for individual lines were correct. Common errors were mostly related to relational operators and branching conditionals (e.g. where Codex stated "less than or equal to x" when the corresponding code was checking "less than x").

3.1.3 Rapid progress. In this section, we described early work in which code explanations were generated using a version of the Codex model that was available in early 2022 (specifically, 'code-davinci-001'). Less than a year later, code explanations generated by models such as ChatGPT are considerably better and more consistently accurate. Figure 4 illustrates one example of a code explanation generated by ChatGPT when provided only the code shown in the "Sample solution" area in Figure 3 and using the same prompt for a line-by-line explanation as described in this section.

The quality of LLM-generated learning resources is likely to continue improving alongside model capabilities. For example, MacNeil et al. found that code explanations generated by the more recent GPT-3 model were consistently more helpful than those generated by Codex [24]. They generated several different kinds of code explanations, deploying them in an online interactive e-book, and found that students reported high-level summaries of code as being more useful for their learning when compared to lower-level detailed explanations of each line. Recent work has also found that LLM-created code explanations are rated more highly on average by students than code explanations created by their peers [18].

We see great potential for LLMs to be applied to the production of a variety of learning resources relevant to computing education. We also expect the quality and accuracy of the generated resources to improve considerably over the near term based on recent trends.



Certainly! Here's a step-by-step explanation of the program:

1. The function `` `count_rebounds` `` takes in a list of players as an argument. Each player is represented as a tuple containing three elements: a player's name (a string), their number of points scored (an integer), and their number of rebounds (also an integer).
2. The function initializes a variable `` `total` `` to 0. This variable will be used to store the total number of rebounds among all the players.
3. The function enters a `` `for` `` loop that iterates over each player in the list of players.
4. For each player, the function adds the player's number of rebounds to the `` `total` `` variable.
5. After the `` `for` `` loop completes, the function returns the value of the `` `total` `` variable. This is the total number of rebounds among all the players.

Figure 4: Explanation generated by the ChatGPT model of the code shown in the 'Sample solution' area in Figure 3.

3.2 Better Programming Error Messages

For over six decades, researchers have identified poor Programming Error Messages (PEMs) as problematic, and significant work remains in this area. Recent work has attempted to put error messages into more natural language by focusing on readability, which has been shown to improve student understanding of error messages and the number of successful code corrections [8]. While it is clear that increasing the readability of PEMs is helpful to novices, doing so at scale, and across languages, remains a challenge.

Leinonen et al. explored the potential of LLMs for improving PEMs [19]. They collected Python error messages that had been reported as most unreadable in prior work and generated code examples that produced these error messages. They prompted the

Codex API with both the code and error message in order to generate explanations of the PEMs and actionable fixes. They found that most of the explanations created by Codex were comprehensible, and that Codex produced an output with an explanation for most inputs.

More recent work has extended this approach by implementing GPT-enhanced LLM explanations of PEMs directly into compilers or automated assessment tools. Taylor et al. deployed GPT explanations to a C/C++ compiler in CS1 and CS2 courses and found it provided accurate explanations in 90% of cases for compile-time errors and 75% of cases for run-time errors [37]. Wang et al. found that students receiving GPT-enhanced PEMs in a large-scale introductory programming course repeated an error 23.5% less often and resolved an error in 36.1% fewer attempts [40]. While there is still work to be done before the decades-old problem is solved, the potential to demystify PEMs in this way is an exciting opportunity only recently made possible.

3.3 Exemplar Solutions

Students often seek exemplar solutions when coding, either to check against their own code or to get help when struggling. However, instructors may not have the time to provide model solutions for every exercise, including historical test and exam questions. AI-generated code offers a time-saving alternative, with the ability to produce a variety of solutions which can help students understand and appreciate different trade-offs in problem-solving, as suggested by Thompson et al. [38].

The ability to generate exemplar solutions automatically can shift the emphasis from just ensuring that code is correct to focusing on the differences between multiple correct solutions, and the need to make judgments on code style and quality. Extensive research on the benefits of peer review of code [12], suggests that it is beneficial to consider multiple solutions to a problem, even if some of them are flawed. Code generation models can be used to create solutions of varying quality, and these can be used for assessment tasks that require students to apply the critical analysis skills needed for code evaluation. This can facilitate discussions about different approaches and the quality of solutions, and provide opportunities for refactoring exercises [10].

3.4 New Pedagogical Approaches

Computing educators are still working through the implications of LLMs in their classrooms and a consensus about how to update pedagogy has yet to form. However, some early approaches are emerging.

3.4.1 LLMs early. In a traditional CS1 course, the initial focus usually begins with syntax and basic programming principles, and it can take time for students to become proficient in these fundamentals. One novel approach for progressing more rapidly to complex problems is to teach students how to use LLMs to handle low-level implementation details. This is exemplified by the approach in the textbook by Zingaro and Porter, "Learn AI-Assisted Python Programming: With GitHub Copilot and ChatGPT" [28]. Students are introduced to the GitHub Copilot plugin within the Visual Studio Code IDE before they have learned to write a single line of Python code. A top-down approach is followed, where students decompose larger projects into smaller functions that are then solvable using Copilot by providing natural language comments. This textbook provides a blueprint for how introductory courses could initially concentrate more on problem-solving and algorithms, rely on automatic code generation for implementation, and defer in-depth and nuanced discussions of syntax until later.

3.4.2 Explaining algorithmic concepts clearly. It is well known that the outputs produced by large language models are very sensitive to their inputs [32]. In fact, "prompt engineering," where effective prompts are crafted, has emerged as a distinct (and nascent) skill when working with these models. 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 [36]. Denny et al. found that prompt engineering strategies, which described algorithmic steps, were effective for solving programming tasks for which Copilot initially generated solutions that were incorrect [6]. Other recent work has shown that developers are more successful working with Copilot when they decompose larger programming statements into smaller tasks and then explicitly prompt Copilot for each of the sub-tasks [1, 13]. It is likely that students will need to develop new skills to communicate effectively with these models. A key skill will be the ability to describe the computational steps they wish to achieve in natural language as a way of guiding the model to produce valid outputs.

3.4.3 Specification-focused tasks. One way for students to learn how to create effective prompts is to focus on writing task specifications. In a traditional introductory course, novices are presented with problem statements that have been very carefully specified by the instructor to be clear and unambiguous. Such detailed specifications provide excellent context for code generation models to produce correct code solutions. New types of problems could task students with generating clear specifications themselves, and thus strengthen skills around LLM prompting. For example, this is the goal of 'prompt problems' [7], in which students are presented with a visual representation of a problem that illustrates how input values should be transformed to an output. Their task is to devise a prompt that would guide an LLM to generate the code required to solve the problem. Prompt-generated code is evaluated automatically and can be refined iteratively until it successfully solves the problem. Recent work investigating classroom use of prompt problems has shown that students find them useful for strengthening their computational thinking skills and exposing them to new programming constructs.

3.4.4 A focus on refactoring. Students sometimes experience difficulty getting started on programming assignments, sometimes referred to as the programmer's writer's block. Recent work found that Copilot can help students overcome this barrier by immediately providing starter code, enabling them to build upon existing code rather than starting from scratch with a blank code editor [39]. This approach may require a shift in focus towards tasks such as rewriting, refactoring, and debugging code, but it provides the opportunity to help students maintain momentum in a realistic setting where the ability to evaluate, rewrite, and extend code is often more important than writing every line of code from scratch.

3.5 Designing LLM Tools

Programmers around the world, not just novices, will be utilizing code generators in an increasing capacity moving forward. Exploring the integration of LLMs directly into educational environments, such as auto-graders and online textbooks, will be an important area of research moving forward. There is a need in such environments for appropriate guardrails so that generated outputs usefully support learning, without immediately revealing solutions or overwhelming novices with the complexity or quantity of feedback. Indeed, the announcement of GPT-4[11] highlighted the example of a 'Socratic tutor' that would respond to a student's requests with probing questions rather than revealing answers directly. One example of this integration in computing education is the work of Liffiton et al. on CodeHelp, an LLM-powered tool that uses prompt-based guardrails to provide programming students with real-time help but without directly revealing code solutions [22].

In general, adapting the feedback generated by LLMs to maximize learning in educational environments is likely to be an important research focus in the near future. Concrete recommendations are already beginning to emerge from very recent work in this space. First, the over-utilization of code generators by novices will generally decrease the number of errors they see. This seems like a positive experience, though it appears they are ill-equipped to deal with the errors they do see when presented with them [14]. This means that tools must be designed to help users (of all skill levels) through the error-feedback loop. Second, generating and inserting large blocks of code may be counter-productive for users at all levels. This requires users to read through code they did not write, sometimes at a more sophisticated level than they are familiar with. Novices may be intimidated by such code generation [14] or may spend too much time reading code that does not further their goals [30]. Therefore, AI code generators should include a way for users to control the amount of code insertion and to specify how to step through a multi-part segment of generated code. Third, the fact that AI code generators are black boxes means that programmers of all skill levels may struggle to create correct mental models of how they work, which could harm their ability to fully utilize them or learn from their outputs. Explainable AI (XAI) patterns could be helpful here, such as exposing to the user a confidence value and user skill estimation above the generated code suggestion [30]. These three suggestions are only the beginning of a long line of research on how to helpfully design usable AI code generators that empower novice learners and enhance programmer productivity.

4 WHERE DO WE GO FROM HERE?

The emergence of powerful code generation models has led to speculation about the future of the computing discipline. In a recent CACM viewpoint article, Welsh claims they herald the "end of programming" and believes there is major upheaval ahead for which few are prepared, as the vast majority of classic computer science will become irrelevant [41]. In an even more recent article on BLOG@CACM, Meyer is equally impressed by the breakthroughs, placing them alongside the World Wide Web and object-oriented programming as a once-in-a-generation technology, but also takes a more optimistic view[12]. In fact, Meyer predicts a resurgence in the need for classic software engineering skills such as requirements analysis, formulating precise specifications, and software verification.

Although the impact of generative AI tools is already evident for software developers, the long-term changes for computing education are less clear. Experts appreciate this new technology only because they already understand the underlying fundamentals. The ability to quickly generate large amounts of code does not eliminate the need to understand, modify, and debug code, but instead, it highlights how important it is to develop these basic competencies! Code literacy skills are essential in order to critically analyze what is being produced to ensure alignment between one's intentions and the generated code. Without the skills to read, test, and verify that code does what is intended, users risk becoming mere consumers of the generated content, relying on blind faith more than developed expertise. We argue that writing code remains a valuable way for novices to learn the fundamental concepts essential for code literacy.

Although professional developers may indeed spend less time in the future writing 'low-level' code, we believe that generated code will still need to be modified and integrated into larger programs. Although we do expect to see some shift in emphasis, even in introductory courses, towards modifying code generated by AI tools, the ability to edit such outputs and compose code in today's high-level languages will likely remain a fundamental skill for students of computing. This aligns with Yellin's recent viewpoint that as programs increase in complexity, natural language becomes too imprecise an instrument with which to specify them [42]. At some point, editing code directly is more effective than issuing clarifying instructions in natural language.

Tools like Copilot and ChatGPT, harnessed correctly, have the potential to be valuable assistants for this learning. We see these tools as serving a valuable teaching support role, used to explain concepts to a broad and diverse range of learners, generate exemplar code to illustrate those concepts, and generate useful learning resources that are contextualized to the interests of individuals. We also anticipate the emergence of new pedagogies that leverage code generation tools, including explicit teaching of effective ways to communicate with the tools, and tasks that focus on problem specification rather than implementation.

In light of the rapid adoption of generative AI tools, it is essential that educators evolve their teaching methods and approaches to assessment. Curricula should also expand to cover the broader societal impact of generative AI, including pertinent legal, ethical and economic issues. We believe it is imperative to get ahead of the use of these tools, incorporate them into our classrooms from the very beginning, and teach students to use them responsibly. In short, we must adopt or perish. Adoption, in this case, is not just a necessity – it's an opportunity for revitalization.

REFERENCES

[1] 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

[11] openai.com/research/gpt-4

[12] cacm.acm.org/blogs/blog-cacm/268103-what-do-chatgpt-and-ai-based-automatic-program-generation-mean-for-the-future-of-software/fulltext

[2] Brett A. Becker, James Prather, Paul Denny, Andrew Luxton-Reilly, James Finnie-Ansley, et al. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities And Challenges of AI Code Generation. In *Proceedings of the 54th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '23)*. Association for Computing Machinery.

[3] Stella Biderman and Edward Raff. 2022. Fooling MOSS Detection with Pretrained Language Models. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 2933–2943. https://doi.org/10.1145/3511808.3557079

[4] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, et al. 2021. On the Opportunities and Risks of Foundation Models. https://doi.org/10.48550/ARXIV.2108.07258

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

[6] 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)*. Association for Computing Machinery, New York, NY, USA, 1136–1142. https://doi.org/10.1145/3545945.3569823

[7] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, et al. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 7. https://doi.org/10.1145/3626252.3630909

[8] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, et al. 2021. On Designing Programming Error Messages for Novices: Readability and Its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY,USA, Article 55, 15 pages. https://doi.org/10.1145/3411764.3445696

[9] Thomas Dohmke. 2023. GitHub Copilot for Business is now available - The GitHub Blog. https://github.blog/2023-02-14-github-copilot-for-business-is-now-available/. (Accessed on 11/16/2023).

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

[11] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, et al. 2023. My AI Wants to Know If This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *Proceedings of the 25th Australasian Computing Education Conference (ACE '23)*. Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/3576123.3576134

[12] 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

[13] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, et al. 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 (CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 386, 19 pages. https://doi.org/10.1145/3491102.3501870

[14] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, et al. 2023. Studying the Effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 455, 23 pages. https://doi.org/10.1145/3544548.3580919

[15] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara J. Ericson, David Weintrop, et al. 2023. How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. In *Proceedings of the 23rd Koli Calling Conference on Computing Education Research (Koli Calling '23)*. 10.

[16] Clifton Kussmaul. 2012. Process Oriented Guided Inquiry Learning (POGIL) for Computer Science. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 373–378. https://doi.org/10.1145/2157136.2157246

[17] Sam Lau and Philip Guo. 2023. From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools Such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1 (ICER '23)*. Association for Computing Machinery, New York, NY, USA, 106–121. https://doi.org/10.1145/3568813.3600138

[18] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, et al. 2023. Comparing Code Explanations Created by Students and Large Language Models. https://doi.org/10.48550/arXiv.2304.03938

[19] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, et al. 2023. Using Large Language Models to Enhance Programming Error Messages. In *Proceedings of the 2023 ACM SIGCSE Technical Symposium on Computer Science Education*. https://doi.org/10.1145/3545945.3569770

[20] Hang Li. 2022. Language Models: Past, Present, and Future. *Commun. ACM* 65, 7 (Jul 2022), 56–63. https://doi.org/10.1145/3490443

[21] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[22] Mark Liffiton, Brad Sheese, Jaromir Savelka, and Paul Denny. 2023. CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. arXiv:cs.CY/2308.06921

[23] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, et al. 2022. Metacognition and Self-Regulation in Programming Education: Theories and Exemplars of Use. *ACM Trans. Comput. Educ.* 22, 4, Article 39 (sep 2022), 31 pages. https://doi.org/10.1145/3487050

[24] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, et al. 2023. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 931–937. https://doi.org/10.1145/3545945.3569785

[25] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP46214.2022.9833571

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

[27] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? (2022). https://doi.org/10.48550/arXiv.2211.03622

[28] Leo Porter and Daniel Zingaro. 2023. *Learn AI-Assisted Python Programming With GitHub Copilot and ChatGPT.* Manning, Shelter Island, NY, USA. https://www.manning.com/books/learn-ai-assisted-python-programming

[29] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, et al. 2023. The Robots are Here: Navigating the Generative AI Revolution in Computing Education. arXiv:cs.CY/2310.00658

[30] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, et al. 2023. "It's Weird That It Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Trans. Comput.-Hum. Interact.* (aug 2023). https://doi.org/10.1145/3617367 Just Accepted.

[31] Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A. Becker, et al. 2023. Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 299–305. https://doi.org/10.1145/3587102.3588805

[32] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems (CHI EA '21)*. Association for Computing Machinery, New York, NY, USA, Article 314, 7 pages. https://doi.org/10.1145/3411763.3451760

[33] Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, et al. 2022. What is it like to program with artificial intelligence? arXiv:cs.HC/2208.06213

[34] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1 (ICER '22)*. Association for Computing Machinery, NY NY, USA, 27–43. https://doi.org/10.1145/3501385.3543957

[35] Simon, Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, et al. 2016. Negotiating the Maze of Academic Integrity in Computing Education. In *Proceedings of the 2016 ITiCSE Working Group Reports (ITiCSE '16)*. Association for Computing Machinery, NY NY, USA, 57–80. https://doi.org/10.1145/3024906.3024910

[36] Leonard Tang, Elizabeth Ke, Nikhil Singh, Bo Feng, Derek Austin, et al. 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, 612–615.

[37] Andrew Taylor, Alexandra Vassar, Jake Renzella, and Hammond Pearce. 2023. Dcc –help: Generating Context-Aware Compiler Error Explanations with Large Language Models. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 7.

[38] Errol Thompson, Jacqueline Whalley, RF Lister, and Beth Simon. 2006. Code Classification as a Learning and Assessment Exercise for Novice Programmers. In *19th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2006)*. National Advisory Comittee on Computing Qualifications, Wellington, New Zealand, 291–298.

[39] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. Association for Computing Machinery, NY NY, USA, 1–7.

[40] Sierra Wang, Chris Piech, and John C. Mitchell. 2023. A Large Scale RCT on Effective Error Messages in CS1. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 7.

[41] Matt Welsh. 2023. The End of Programming. *Commun. ACM* 66, 1 (Jan 2023), 34–35. https://doi.org/10.1145/3570220

[42] Daniel M. Yellin. 2023. The Premature Obituary of Programming. *Commun. ACM* 66, 2 (Feb 2023), 41–44. https://doi.org/10.1145/3555367

[43] Craig B. Zilles, Matthew West, Geoffrey L. Herman, and Timothy Bretl. 2019. Every University Should Have a Computer-Based Testing Facility. In *Proceedings of the 11th International Conference on Computer Supported Education, CSEDU 2019, Heraklion, Crete, Greece, May 2-4, 2019, Volume 1*, H. Chad Lane, Susan Zvacek, and James Uhomoibhi (Eds.). SciTePress, 414–420. https://doi.org/10.5220/0007753304140420