



Language-Model Assisted Learning How to Program?

Jochen L. Leidner^{1,2}(✉)  and Michael Reiche¹

¹ Coburg University of Applied Sciences, Friedrich Streib-Straße 2, 96450 Coburg, Germany

² University of Sheffield, Regents Court, 211 Portobello, Sheffield S1 4DP, UK
leidner@acm.org

Abstract. Foundational language models have forever changed how NLP prototypes may be rapidly constructed, dramatically reducing the “cost of curiosity”. This also affects the way we can teach and learn how to program.

In this paper, we explore how well foundational models such as large, pre-trained neural transformers can answer questions pertaining to programming in a “learning to code” context: we present a new dataset comprising questions that students that learn how to program and students of particular programming languages – as offered by typical undergraduate university courses – typically ask. We cover both fundamental concepts in programming and also specific programming language issues. Although our study focuses on English, we believe results would be similar for other human languages due to the multilingual nature of many foundational language models. We explore how well a foundational (generic) pre-trained language model can answer them.

To the best of our knowledge, this is one of the first studies that assesses *how well* generic foundational models and applications like ChatGPT are capable of answering different types of typical programming-related questions. This is a question of primary importance if we consider using such models to assist human students in their struggle to become (good) programmers.

Keywords: Computer-Supported Instruction · AI for Teaching · Improving the Teaching of AI · Pretrained Foundational Models · Education

1 Introduction

1.1 Background and Motivation

Learning how to program is an essential part of studying computer science, because it enables students to implement systems themselves that embody their own (and later, their clients’) ideas. It is also an essential and valuable transferable skill: students of other subjects, from architecture to physics, are also often expected to take programming classes, and software developers are above-average earners.

Nevertheless, like mastering to play the piano or mastering a foreign language, learning to program entails numerous challenges: to grasp the nature of the problem domain (get the necessary background), to understand a specific problem that a computer program to be written is expected to solve, to decompose the problem, to master the lexis (keywords and operators), syntax (EBNF grammar), semantics (e.g. types), development environment (IDEs or command line, run-time) and libraries of the programming language(s) used, to design the architecture of the program that is the tentative solution, to implement the code by making informed choices between self-coded parts and parts re-used by calling external third-party libraries, commercial or open-source, to handle edge cases and errors, to test the code, to document the code. Therefore, it is hardly surprising that three-year Bachelor programs do not produce experienced programmers, given that programming is only one of the many skills/knowledge areas of the computer science curriculum.

In the past, many efforts have gone into the better or faster teaching of programming by supporting the human learner with computational help, including the use of AI techniques (planning the progress, modeling the learner’s grasp of concepts and his/her progress).

Very recently, the introduction of neural transformers and large pre-trained language models (also called “foundational models”), which are trained with general-purpose *un-annotated* human prose language, and sometimes code fragments, has dramatically changes the way natural language applications can be prototyped. Systems like Google’s BERT [5], OpenAI’s GPT-3, GPT-3.5, GPT-4 and the application ChatGPT [12,13], and many others based on deep neural networks featuring the transformer architecture [18] permit to directly pose a question in English and other languages and get an answer back, also in a human language. Although these models were originally intended to be “just” generative models of *language production*, the fact that they were trained with vast quantities of text, including terabytes of World Wide Web content, means that the language material to train the systems also articulated an enormous amount of *world knowledge*, thus implicitly solving the knowledge bottleneck challenge that prevented progress in AI in the 1980 s.

1.2 Research Question

In this paper, we explore the research question “*How well can a (generic) neural transformer model answer programming questions?*”. It is important to know to what degree pre-trained models can cover answers to such questions, especially as they were not originally designed to provide knowledge (they are language models) and also as they were not *a priori* designed as programming aids (again, they are (human) language models). This research question is distinct from the—more interesting but harder-to-answer—question “*How well can one (learn how to) program when relying (only) on a foundational language model?*”.)

To this end, we have manually collected a set of questions from the “learning how to program” domain; while they are not real questions collected from students, they are informed by decades of programming and teaching how to

program between the authors, and they are therefore indicative of the nature of questions real students would ask (and have repeatedly asked over the years). Specifically, to what extent can a pre-trained language model such as a neural transformer like ChatGPT provide (1.) code answers or (2.) answers about code that are (i.) correct and that (ii.) do not contain dangerous omissions (e.g. leaving out error handling) or misleading output (foundational models are known to “hallucinate”, which means providing untrue output as these models have no notions of truth or falsehood, as they focus on how to say something well).

2 Related Work

Early Computer-Based Initiatives to Support Students. After Carbonell’s early and seminal work on intelligent tutoring systems [4], the late 1970s the 1980s saw a number of different approaches, including those using traditional AI methods: *BIP-I*, *BIP-II* (basic programming; Barr et al., 1976), *BRIDGE* (programming; Bonar 1985); *Flow Tutor* (FLOW programming language, Genter, 1977), *LISP Tutor* (LISP programming; Anderson and Reiser, 1985); *MALT* (basic machine language programming; Koffman and Blount, 1975); *MENO-Tutor* (basic Pascal programming; Woolf and McDonald, 1984); *PROUST* (Pascal programming; Soloway and Johnson, 1984); *SCENT-3 Advisor* (McCalla et al., 1988); *SPADE* (basic LOGO programming; Goldstein and Miller, 1976); and *TALUS* (basic LISP programming, Murray, 1987).

Robins, Rountree and Rountree review work on teaching and learning programming [11]. Koulouri, Lauria and Macredie [7] evaluate quantitatively alternative approaches to teaching beginners how to program.

Foundational Neural Language Models. OpenAI’s GPT-3 [3] and ChatGPT [12] have been early foundational models that have been transformational in natural language processing: they showed how large, pre-trained language models such as neural transformers can dramatically reduce the development time of NLP systems by using large quantities of un-annotated text to train general-purpose “foundational” models. Our experiments use OpenAI’s ChatGPT model.

Foundational Models and Programming. Microsoft’s *GitHub Copilot* (based on Open AI Inc.’s Codex model¹) was the first language model aimed at helping coders that was deployed at large-scale (on the Web-based source code revision control service Github.com). [17] describe a human experiment comprising 24 students that use Copilot for three programming tasks and its impact on task completion time and success rate. [1] report on an analysis of how 20 programmers interacted with Copilot. They observed that behavior could be grouped into two modes, acceleration mode, where a programmer uses Copilot to complete the code faster and exploration mode, where a programmer uses Copilot to explore various alternative options for solving a coding problem. [15] report on a Microsoft study that aimed to use a generic neural transformer model to extract

¹ see <https://openai.com/blog/openai-codex>.

information about locking, exceptions and performance from natural language comments of a large software repository. Bird et al. [2] also describe a case study where a set of subjects got instructed how to use Copilot, and then were given two tasks, namely to create a Tic Tac Toe game and to write code that sends an email programmatically via a Sendmail API. The authors describe how subjects responses to questions indicate an increase in productivity. In 2022, Imai, when studying human-computer “pair” programming found that programming with Copilot helps generate more lines of code (LoC) than human pair-programming in the same period of time, but at a lower quality level [6]. Surameery and Shakor provide a high-level comparison of debugging using Chat GPT versus using traditional debugging tools, and conclude that foundational language models can provide a useful expansion of the debugging toolbox of the future by providing bug prediction and bug explanation capabilities [16]. Sarsa et al. [14] present a very interesting approach: they explore how well foundational LMs can *generate programming exercises*. In a sense, this is the inverse exercise of our RQ1, which explores their ability to *answer* (human-provided) questions.² In the context of a next-generation programming education e-book, the same group investigated LMs’ power to explain code in an educational context [9]; they let human students rate the usefulness of the automated output.³ Leinonen et al. also compare code *explanations* created by human students and automatic large language models [8]. They look for differences in accuracy between students and LMs; in contrast, we explore the absolute correctness of human questions against LM answers (as evaluated by a human *expert*).

None of these works uses expert judgment to score a LM’s ability to answer coding questions based on an open corpus.

3 Scope

We collected a set of questions based on the author’s experience in using (from Scheme over C/C++ to Rust) and teaching (from FORTRAN 90 over Java to Python) various programming languages included general questions of understanding the programming process (c.f. Table 3) as well as questions in or about specific programming languages (c.f. Table 4). To mitigate the problem of personal bias, we checked the programmer help Website [StackExchange.org](https://stackoverflow.com) for the number of times similar questions have been asked, to ensure that at least for a sizeable subset of questions, we have evidence that they really already occurred (Table 2).

We selected programming concept questions based on the typical topics that create difficulties (recursion, type systems etc.), and we selected programming languages that are important enough (leaving out many others e.g. AWK, FORTH, Erlang) and familiar to the author (leaving out e.g. BCPL, Verilog, Wolfram language, BLISS and Snobol).

² We thank an anonymous reviewer for making us aware of the work of Sarsa, MacNeil, Leinonen and co-workers.

³ Questions for code explanations are one of our 4 question types: see type 3 in Table 3.

Table 1. A Sample of Programming Concepts Covered in the Dataset

abstraction	ACID	anonymous function
divide and conquer	domain-specific language	generator function
immutable object	linked list	mailbox
recursion	REST	reusability
singleton class	save data to a file	window

Table 2. A List of Error Types Covered in the Dataset

syntax error	wrong type	lifetime error
scope error	logical error	forgot to initialize
unallocated memory	index out of bounds	null pointer error (

4 Method

We execute the set of questions against the OpenAI Inc. ChatGPT API, one at a time. To implement the processing by the language model, we used a bash script, which sends questions to ChatGPT via the `sgpt` command⁴ and stores the response in an SQL database. Our question dataset was processed on a MacBook Air 10 (2021) with ARM M1 processor in 12:26 min including network round-trip time.

5 Dataset

The resulting questions together with the answers provided by the ChatGPT model and the metadata described in Appendix B is available from our GitHub repository⁵ and, at the time of writing, comprises $N = 105$ questions, model responses (as of July 20, 2023, using the May 23 version of the model) and metadata. Tables 4 and 1 provide the number of questions per concept and language in parentheses.

6 Towards an Evaluation

6.1 Quantitative Evaluation

Although we will also provide numbers, our overall evaluation approach is qualitative; due to the small size of our corpus, our numbers are dominated by the small number of examples of each of the many phenomena that should be studied. Nevertheless, as we shall see, a consistent pattern emerged.

⁴ see <https://github.com/tbckr/sgpt>.

⁵ see <https://github.com/Information-Access-Research-Group-IARG/Programming-Questions-for-ChatGPT>.

Table 3. General Question Types Covered in the Dataset

No	Question Type	Example
1	Syntax	<i>How do I formulate X in language Y</i>
2	Idiom	<i>How do I achieve to get X done in language Y</i>
3	Explain	<i>What does code X do?</i>
4	Diagnose	<i>Why does code X not work?</i> <i>Why does code X not do Y?</i>

Table 4. Programming Languages Covered in the Dataset

Python	C++	Java	C
JavaScript	Julia	Kotlin	Bash/sh
Prolog	Scheme/LISP	Rust	FORTRAN
COBOL	R	Matlab	

We manually graded the answers of the ChatGPT model for all questions, assigning an ordinal rating of 0 for wrong/incorrect responses if *any* syntactic errors, semantic errors, type errors, or logical errors were contained in the response, or if code shown did not implement what was requested. Partially correct answers that were helpful (a rather subjective notion, admittedly) were assigned a rating of 1, and fully correct and relevant answers were assigned a rating of 2. We added the verbatim output and grade to the database.

This way of scoring is admittedly crude, and in future work, more detailed aspect-oriented grading experiments should be carried out. However, our three-point ordinal scoring method worked well enough on the questions we had to swiftly grade the set of questions gathered, and to get a sense of ChatGPT’s abilities.

Results. Based on our scoring method, the 105 questions reached 158 points, leading to an arithmetic mean of $\text{Avg} = 1.7753$. Most questions get at least partial credit: only one question was marked 0, because the question about anonymous functions in Kotlin led to an explanation for named functions instead.

This is work in progress; we plan to extend the database with material found to be recurring in student programming classes, and to provide a more extensive evaluation. One could also collect realistic questions from students and validate questions against StackExchange regarding their frequency of occurrence.

6.2 Qualitative Evaluation

How useful are the answers? They are often impeccable. For example:

► Human Question I:

What is divide and conquer? Explain it to me using a chess game as an example.

► Machine Answer I:

Divide and conquer is a programming paradigm that involves breaking down a problem into smaller subproblems, solving them independently, and then combining the solutions to solve the original problem. It is often used to solve complex problems more efficiently. In the domain of writing a chess game, an example of using divide and conquer could be implementing the move validation logic. Instead of checking the validity of a move for the entire chessboard, the board can be divided into smaller sections, such as rows or columns. Each section can then be validated independently, reducing the complexity of the problem. For instance, when validating a move for a specific piece, the program can divide the chessboard into rows and check if the move is valid within that row. Similarly, it can divide the board into columns and validate the move within each column. By breaking down the problem into smaller sections, the move validation process becomes more manageable and efficient.

► Human Question II:

What is the difference between `String` and `StringBuffer` in Java?

► Machine Answer II:

The main difference between `StringBuffer` and `String` in Java is that `StringBuffer` is mutable, meaning its value can be changed, while `String` is immutable, meaning its value cannot be changed once it is created. In other words, when you modify a `StringBuffer` object, it does not create a new object, but instead modifies the existing object. This can be useful when you need to perform a lot of string manipulation operations, as it avoids creating multiple new objects. On the other hand, when you modify a `String` object, it creates a new object with the modified value, leaving the original object unchanged. This can be less efficient when performing a lot of string manipulation operations, as it involves creating multiple new objects. Another difference is that `StringBuffer` is synchronized, which means it is thread-safe and can be used in multi-threaded environments without any issues. `String`, on the other hand, is not synchronized. In summary, if you need to perform a lot of string manipulation operations or if you are working in a multi-threaded environment, `StringBuffer` is a better choice. If you don't need to modify the string and want to ensure immutability, `String` is the way to go.

On the other hand, sometimes functional but incomplete code is returned, as in the following:

► Machine Answer III:

You can load a file containing whitespace-separated integers into an array in Python by following these steps:

1. Open the file using the `open()` function and specify the file path and mode. For example, to open a file named `‘‘numbers.txt’’` in read mode, you can use: `(...) file = open(‘‘numbers.txt’’, ‘‘r’’) (...)`

Results. The code in the response is working and well-explained, but a hypothetical “file not found” situation is not handled. However, to be fair, many online Website also lack this level of sophistication.

7 Discussion

7.1 Accomplishments

We reported on an ongoing project to collect questions of the kind that learners of programming concepts and common programming languages may face. Our findings suggest foundational language models are capable of solving programming tasks at the level of a good to very good student, which is remarkable, in particular due to the fact that ChatGPT was not specifically developed as a programming assistant (unlike Copilot).

7.2 Limitations

Our work is still small scale, and our sample suffers from selection bias. We anticipate that a Wizard of Oz experiment with real students could lead to a bigger and better corpus, and well supplement our collection effort. We need to increase the percentage of questions that explain code and that contain bugs in this process. Our work is also limited in that we have not yet conducted any form of inter-coder agreement. Another limitation is that in some countries (e.g. Germany), the student have a right to be taught the correct solutions, so it is not acceptable for e.g. a chatbot to occasionally get the answer wrong (“hallucination”); this could be addressed by warnings to the user. In the box in Table 5, we report on a parallel experiment in which students without programming skills were able to solve a technical assignment assisted by ChatGPT.

However, preliminary experiments by the second author have shown that while task completion probability and task completion time improve when supporting students with a chat-enabled transformer, understanding of programming concepts does not (see Box “A Teaching Experiment” in Table 5).

8 Ethical Reflections

The ability of language models historically came as a surprise: emerging out of the research into large (human) language models that got pre-trained with vast amounts of text, data crawled from the World Wide Web included not just plenty of useful text, but also code repositories, programming discussion forums

Table 5. A Case Study with $N = 2$ Teams of Non-Programmers

In the summer term of 2023, we conducted a controlled experiment “Artificial Intelligence Team Project” at Coburg University of Applied Sciences, with 6 Master’s students in business studies. The students were grouped into à 3 students each. The task was to classify an annotated dataset of about 400k German-language feedback messages and numerical ratings of patients’ experiences with medical practitioners according to the sentiment polarity expressed (into positive and negative comments) using a simple, two-class logistic regression model. To achieve this, Python code had to be used for data understanding, data preparation and modelling. One question was of how well the students, absent any machine learning experience and hardly any programming experience, could generate functional code fragments and attach them to each other using only a pre-trained language model. It turned out that both groups generated functional Python code, and produced two classifiers with $F1 > 98\%$. OpenAI’s ChatGPT (model 3.5 version of May 24, 2023, used with German conversations) as the pre-trained language model used was largely responsible for the success of the project. Indeed, every line of code was generated by ChatGPT, corrected in case of error, and described for the students. This was achieved by adapting prompts, such as instructing ChatGPT to please answer in understandable language or to please correct the previously generated code using an error from Jupyter Notebook or PyCharm. The prompts for generating programme code were formulated without or with hardly any computer science-specific technical terms. A typical prompt for code generation was (translated from German): “*How do i remove stop words in column 3 in a dataframe?*” A typical prompt for code improvement was the following (again, translated from German): “*Instead of the accuracy, it should be output how many were correctly identified with value 0.*”

Anecdotal Results: All students reported ChatGPT changed the way they worked, it enabled them to work faster, and to bridge gaps in their knowledge. One group also reported, however, that their understanding of programming did not deepen in this exercise using ChatGPT. ■

etc. One challenge is that the exact set of Web sites included in the training of the proprietary models like OpenAI’s ChatGPT remain unpublished.

In any case, this study showed that a model that was *not specifically intended* for this purpose is capable of solving substantial programming sub-tasks. This a case of morally positive unintended use; however, there are also uses that are ethically questionable, such as using a foundational language model for solving exercises when its use is forbidden. It will only be possible to a very limited extent to be able to tell, by humans or machines, whether a foundational model was used in the course of solving a programming exercise. Therefore, if programming exercises are to be graded as part of coursework, either a non-networked environment must be created, or programming has to happen based on pen and paper only (perhaps the latter is less desirable than the former due to its artificial nature, but creating a functional but isolated, secured, non-networked environ-

ment is also a challenge, not to mention the pervasiveness of networked mobile devices).

One fundamental danger is that the use of foundational models for programming will become very common (as it no doubt will), and as a result, safety critical code will be in part originate from auto-generated code that contains only insufficient error handling. This scenario is likely due to company’s incentives to increase profits and reduce cost rather than maximize quality and minimize software defects.

9 Summary, Conclusion and Future Work

Foundational language models were pre-trained with human language material, and in the process ingested substantial source code in various languages, too; as a consequence, they are *de facto* also models of how to program, despite unreliable ones. We found evidence of programming knowledge could be retrieved on a broad set of tasks and programming languages, which may aid beginners and speed up experts.

In this paper, we looked at one generic (foundational) models’ programming abilities, which is a necessary but not sufficient condition for answering the question in this paper’s title; we could answer the “ability” question overall affirmatively. Large pre-trained neural transformers like the one underlying the ChatGPT application encode substantial programming and programming language knowledge, which can be accessed using a convenient interface and in many human languages. Whether and how foundational language models can assist humans in the process of learning how to program, the overarching question, further requires us to find out whether they can help learners perform and deepen learner understanding, which should be explored in future work (see also [10] in this volume).

Further work should explore cross-language consistency (many learners are not English native speakers). A comparison of multiple alternative responses of the language model used would also be interesting.⁶ Using a detailed prompt may further improve the results; our experience with other transformer experiments has shown that the time to try our various prompts, i.e. prefixing the questions with some prose to set a context, often leads to substantial improvements. One approach could be the collection and clustering of (abstract syntax trees of) problem–answer pairs in terms of code in a way that mixes human-originating answers with machine-generated answers so that students can see that a human solution for their question may already exist, so they do not have to rely on (relatively more error-prone) machine suggestions. Finally, a benchmark evaluation that compares an approach that retrieves human forum answers from StackExchange with automatically synthesized answers from language models would be interesting.

⁶ We are grateful to a reviewer for pointing out this idea.

Acknowledgements. We would like to thank our anonymous referees for their valuable feedback, which helped improve the quality of this paper. The research presented in this paper was partially funded by project VoLL-KI (BMBF/German Federal Ministry for Education and Research) under grants 16DHBKI089, 16DHBKI090 and 16DHBKI091, and by an award to the first author under the Hightech Agenda Bavaria; this funding support is gratefully acknowledged. All opinions and errors are solely the authors' and do not necessarily reflect the opinions of any funding agency.

A Some Sample Questions from our Dataset

Show me the C code to start and stop a precise timer for code benchmarking.

In C, how can I generate a random number between 1 and 100?

In C, how can I define a Unicode string?

In C, how can I portably access the elements of the header of a binary file like JPG?

In portable standard ISO C++20, how do you read a *.csv file into RAM without using a library?

In portable standard ISO C++20, how do you read a *.csv file into RAM?

In portable standard ISO C++20, how do you read a *.csv file into RAM using the standard library or an open source library under the MIT, BSD, Apache or LGPL licenses.?

In Python, how can I draw simple graphics using LOGO-like turtle graphics commands?

In Python, how can I professionally render a contingency matrix that shows the performance of a binary classifier?

In Java, how do I iterate over all keys of a hashtable?

In Java, how can I replace all matches of a regular expression?

In C, how can I create a 3-dimensional matrix of float objects that is safe from buffer overflow errors?

In Julia, how can I plot a ROC curve?

In Julia, how can I print a contingency matrix?

In Julia, how can I define a three-dimensional matrix of floats?

In Julia, which library provides an implementation of the Viterbi algorithm for Hidden Markov Models?

Show me the Julia code for generating a random undirected graph.

In Rust, how can I design functions to return errors systematically? In portable standard ISO C++20, how do you read a *.csv file into RAM without using a library?

In portable standard ISO C++20, how do you read a *.csv file into RAM?

In portable standard ISO C++20, how do you read a *.csv file into RAM using the standard library or an open source library under the MIT, BSD, Apache or LGPL licenses.?

In Python, how can I draw simple graphics using LOGO-like turtle graphics commands?

In Python, how can I professionally render a contingency matrix that shows the performance of a binary classifier?

In Java, how can I implement a singleton class?

In C, how can I create a 3-dimensional matrix of float objects that is safe from buffer overflow errors?

In Julia, how can I view the compiled code for a function?

In Rust, how can I design functions to return errors systematically?

In Rust, how do I implement what would be a class in Java or C++?

In Rust, how do I make a struct printable?

Show me a Kotlin class for singly linked lists.

In Rust, how do I implement what would be a class in Java or C++?

In Rust, how do I make a struct printable?

In Rust, which library is best for fast trie lookup in RAM?

In Rust, which library provides efficient B-tree storage on disk?

In Kotlin, how does the minimum code of an Android mobile app look like?

In Kotlin, what is the syntax for anonymous functions?

What does this C++ code do? ...

What does this Python code do? ...

What does the following Python code do? ...

How can I test in Python whether a CUDA GPU is present?

Explain recursion.

Explain the difference between transient and persistent.

In databases, explain the ACID acronym.

Show me an SQL query that computes aggregate statistics about a table.

If I have an SQL table defined by “CREATE TABLE t(...)”, how can I insert a new author only if he or she does not already exist?

In FORTRAN 95, how can I multiply two 2×2 matrices of integers A and B?

Can I separate a function’s declaration from its implementation in FORTRAN?

Can I separate a function’s declaration from its implementation in C++?

Can I separate a function’s declaration from its implementation in Java?

Explain tail recursion to me.

Show me all bugs and deficiencies in this C code: void

```
nb__net_init(void) { nb__init_timers(); nb__net_state =
malloc(sizeof(nb__net_state)); nb__net_state->num_conn = 0; }
```

What is wrong in the following Python code: ... ? Explain all errors or bugs to me.

Show me all errors or bugs in the following C function: ... How does it look like when it is corrected?

Show me a set of C functions for creating a (singly) linked list, inserting data to a linked list, deleting an item from a linked list and freeing a linked list.

B DDL Database Schema

Table 6 shows the Data Definition Language (DDL) specification of the relational database that we use to store and distribute the dataset described in this

paper; we use SQLite, which is simple, fast and already pre-installed on many machines. Each question gets a unique ID, the question string is paired with an answer string (the model’s response/completion string returned), the name of the programming language is given as a string for reasons of simplicity. The question type refers to the earlier table, and the answer type is incorrect (0), 1 (partially correct) or 2 (correct).

Table 6. DDL Database Schema for the dataset.

```
CREATE TABLE questions (
  id          INTEGER PRIMARY KEY AUTOINCREMENT,
  question   VARCHAR NOT NULL,
  answer     VARCHAR,
  prglang    VARCHAR,
  qtype      INTEGER,
  atype      INTEGER
);
```

References

1. Barke, S., James, M.B., Polikarpova, N.: Grounded Copilot: how programmers interact with code-generating models. Unpublished manuscript. [ArXiv.org](https://arxiv.org/abs/2206.15000) preprint server, Cornell University, New York, NY, USA (2022). <https://arxiv.org/abs/2206.15000>
2. Bird, C., et al.: Taking flight with copilot. *Commun. ACM* **66**(6), 56–62 (2023). <https://doi.org/10.1145/3589996>
3. Brown, T., et al.: Language models are few-shot learners. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) *Advances in Neural Information Processing Systems*. vol. 33, pp. 1877–1901. Curran (2020), https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf
4. Carbonell, J.R.: AI in CAI: an artificial-intelligence approach to computer-assisted instruction. *IEEE Trans. Man-Mach. Syst.* **11**(4), 190–202 (1970). <https://doi.org/10.1109/TMMS.1970.299942>
5. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186. ACL, Minneapolis, MN, USA (2019). <https://doi.org/10.18653/v1/N19-1423>
6. Imai, S.: Is GitHub Copilot a substitute for human pair-programming? An empirical study. In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 319–321. ICSE 2022, ACM, New York, NY, USA (2022). <https://doi.org/10.1145/3510454.3522684>

7. Koulouri, T., Lauria, S., Macredie, R.D.: Teaching introductory programming: a quantitative evaluation of different approaches. *ACM Trans. Comput. Educ.* **14**(4), 1–28 (2015). <https://doi.org/10.1145/2662412>
8. Leinonen, J., et al.: Comparing code explanations created by students and large language models (2023). unpublished manuscript, arXiv cs.CY 2304.03938, Cornell University pre-print server
9. MacNeil, S., et al.: 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. vol. 1, pp. 931–937. SIGCSE 2023, Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3545945.3569785>
10. Reiche, M., Leidner, J.: Bridging the programming skill gap with ChatGPT: A machine learning project with business students. In: Nowaczyk et al., S. (ed.) ECAI 2023 Workshops, Kraków, Poland. CCIS, Springer Nature, Cham, Switzerland (2023), Workshop on AI for AI Learning, in this volume
11. Robins, A., Rountree, J., Rountree, N.: Learning and teaching programming: a review and discussion. *Comput. Sci. Educ.* **13**(2), 137–172 (2003)
12. Roumeliotis, K.I., Tselikas, N.D.: ChatGPT and Open-AI models: a preliminary review. *Future Internet* **15**(6), 192 (2023). <https://doi.org/10.3390/fi15060192>, <https://www.mdpi.com/1999-5903/15/6/192>
13. Sanderson, K.: GPT-4 is here: what scientists think. *Nature* **615**(7954), 773 (2023)
14. Sarsa, S., Denny, P., Hellas, A., Leinonen, J.: 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, pp. 27–43. ICER 2022, Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3501385.3543957>
15. Su, Y., Wan, C., Sethi, U., Lu, S., Musuvathi, M., Nath, S.: HotGPT: how to make software documentation more useful with a large language model? In: Proceedings of the 19th Workshop on Hot Topics in Operating Systems, pp. 87–93. HOTOS 2023, Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3593856.3595910>
16. Surameery, N.M.S., Shakor, M.Y.: Use ChatGPT to solve programming bugs. *Int. J. Inf. Technol. Comput. Eng.* **3**(01), 17–22 (2023). <https://doi.org/10.55529/ijitc.31-17.22>, <https://journal.hmjournals.com/index.php/IJITC/article/view/1679>
17. Vaithilingam, P., Zhang, T., Glassman, E.: Expectation vs. experience: evaluating the usability of code generation tools powered by large language models. In: Extended Abstracts of the 2022 Conference on Human Factors in Computing Systems, pp. 1–7 (2022), <https://dl.acm.org/doi/10.1145/3491101.3519665>
18. Vaswani, A., et al.: Attention is all you need. In: Guyon, I., et al. (eds.) *Advances in Neural Information Processing Systems 30–31st Annual Conference on Neural Information Processing Systems*, Long Beach, CA, 4–9 December 2017, pp. 5999–6010. (NIPS 2017), Curran Associates (2018)