An Empirical Evaluation of GitHub Copilot's Code Suggestions

Nhan Nguyen and Sarah Nadi
University of Alberta
Edmonton, AB, Canada
{nhnguyen,nadi}@ualberta.ca

ABSTRACT

GitHub and OpenAI recently launched Copilot, an "AI pair programmer" that utilizes the power of Natural Language Processing, Static Analysis, Code Synthesis, and Artificial Intelligence. Given a natural language description of the target functionality, Copilot can generate corresponding code in several programming languages. In this paper, we perform an empirical study to evaluate the correctness and understandability of Copilot's suggested code. We use 33 LeetCode questions to create queries for Copilot in four different programming languages. We evaluate the correctness of the corresponding 132 Copilot solutions by running LeetCode's provided tests, and evaluate understandability using SonarQube's cyclomatic complexity and cognitive complexity metrics. We find that Copilot's Java suggestions have the highest correctness score (57%) while JavaScript is the lowest (27%). Overall, Copilot's suggestions have low complexity with no notable differences between the programming languages. We also find some potential Copilot shortcomings, such as generating code that can be further simplified and code that relies on undefined helper methods.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; **Reusability**.

KEYWORDS

Program Synthesis, Codex, GitHub Copilot, Empirical Evaluation

ACM Reference Format:
Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528470

1 INTRODUCTION

There has been lots of research dedicated to improving developer productivity through code synthesis, code search, or other types of code recommender systems [2, 4, 20, 25]. Many of these efforts leverage Artificial Intelligence, specifically deep learning techniques [3, 5, 12, 32]. In June 2021, GitHub and OpenAI introduced GitHub Copilot, an "AI pair programmer" for Visual Studio Code, Neovim, and JetBrains IDEs[11]. Powered by the large-scale OpenAI Codex model, which was trained on open-source GitHub code, Copilot can suggest code snippets in different programming languages [11]. While there have been lots of similar research efforts [12, 14, 31], the seamless integration and availability of Copilot, along with GitHub's backup, naturally created a "hype" in the tech world with many developers already using it through its technical preview or awaiting its usage [1, 21]. However, as pointed by GitHub, Copilot's suggestions "may not always work, or even make sense" [11]. Thus, it is important to assess the correctness and quality of Copilot's suggestions to provide better insights into the overall performance of the tool.

This paper contributes an empirical assessment of GitHub Copilot's capabilities. The insights gained from such a study can help developers understand how to best use Copilot as well as provide insights to Copilot's team and other researchers in this area. We specifically focus on Copilot's synthesized code suggestions, given a natural language description, and answer the following research questions: ***RQ1: How correct are Copilot's code suggestions?*** and ***RQ2: How understandable is the code provided by Copilot?***

To evaluate the correctness of Copilot's suggestions, we use LeetCode[18], a question pool website, which provides us with the context needed to create Copilot queries (function name, parameters, input, and output of the function) as well as test cases that we can use to evaluate correctness of the suggestions. LeetCode questions come with test cases in various programming languages, allowing us to assess the correctness of Copilot's suggestions in 4 different languages (Python, Java, JavaScript, and C). While Copilot produces a ranked list of suggestions, we focus on evaluating its first suggested code snippet. To evaluate the understandability of Copilot's suggestions, we use SonarQube[28] to calculate cognitive complexity and cyclomatic complexity, which been shown to positively correlate with code understandability [9].

Overall, we evaluate Copilot on 33 LeetCode questions in four different languages, totaling 132 queries. Out of Copilot's 132 suggestions, our results show that Java suggestions have the highest likelihood to be correct. Specifically, 57% of Copilot's first Java suggestions pass all the test cases from LeetCode. On the other hand, JavaScript suggestions have the lowest correctness score (27%) and C suggestions have the highest rate of compilation errors (24%). In many of the remaining cases, Copilot suggests a partially correct solution that passes some of the test cases. Overall, we find that Copilot will help the developer completely solve their programming task or provide them with a useful point in 61%-91% of the time, depending on the language.

In terms of understandability, we evaluate Copilot's solutions for only three of the languages, Java, JavaScript, and Python (total of 99 solutions). We find that Copilot's suggestions have low cyclomatic and cognitive complexity (median 5 and 6, respectively), and

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528470

```
1  # https://leetcode.com/problems/longest-increasing-path-in-a-matrix/
2  #Given an m x n integers matrix, return the length of the longest
   increasing path in matrix.
3  #From each cell, you can either move in four directions: left,
   right, up, or down.
4  # You may not move diagonally or move outside the boundary (i.e.,
   wrap-around is not allowed).
5
6  def longestIncreasingPath(self,matrix):
7      Next (Alt+]) Previous (Alt+[) Accept (Tab) Open GitHub Copilot (Ctrl+Enter)
8      if not matrix:
```

(a) Copilot query context

```
1  Synthesizing 10/10 solutions
2
3  =======
4
   Accept Solution
5  def longestIncreasingPath(self, matrix):
6          """
7          :type matrix: List[List[int]]
8          :rtype: int
9          """
10         if not matrix:
11             return 0
12         m, n = len(matrix), len(matrix[0])
13         dp = [[0] * n for _ in range(m)]
14         for i in range(m):
15             for j in range(n):
16                 dp[i][j] = self.dfs(matrix, m, n, i, j, dp)
17         return max(map(max, dp))
18
19 =======
20
   Accept Solution
21 def longestIncreasingPath(self, matrix):
22         """
23         :type matrix: List[List[int]]
```

(b) Copilot complete function suggestions

Figure 1: Example of Copilot query context and suggestions

that there is no statistically significant difference between these languages. We share all data, scripts, and results of our study [23].

2 BACKGROUND AND TERMINOLOGY

GitHub Copilot. Copilot currently offers three main functionalities: convert comments to code, suggest tests matching implementation code, and autofill for repetitive code [11]. We focus on the first functionality, converting comments to code, which is triggered by a user writing a comment to describe the logic they want to implement [11]. While providing only a natural language comment is enough to trigger a Copilot suggestion, it is recommended that users provide meaningful names for functions parameters as well as descriptive comments to get meaningful recommendations [11]. We refer to the combination of natural language comment, function name, and function parameters as the *query context*.

Figure 1a shows an example of using Copilot. Lines 2 to 7 contain the query context, consisting of the function name, parameters, and a natural language problem description. On Line 8, Copilot shows its suggestion for the next line of code. Users can also view suggestions for the entire function by clicking Ctrl + Enter where Copilot will display a number of different suggestions, as shown in Figure 1b.

LeetCode. LeetCode is a popular Question Pool website (QP)[30]. Such websites provide various coding questions on different topics (array, algorithm, sorting, etc) along with corresponding tests to check correctness. Most importantly for our empirical study, each of these LeetCode coding questions contains the relevant information to compose good query contexts for Copilot, following GiHub's recommendations described above. LeetCode also provides a publicly available API to fetch submission details. [10, 13, 26].

Figure 2a shows an example LeetCode question, named Longest Increasing Path in a Matrix[16]. The question contains information like the input (m x n integers matrix), the expected output (the

329. Longest Increasing Path in a Matrix

Hard 4479 77 Add to List Share

Given an `m x n` integers `matrix`, return *the length of the longest increasing path in* `matrix`.

From each cell, you can either move in four directions: left, right, up, or down. You **may not** move **diagonally** or move **outside the boundary** (i.e., wrap-around is not allowed).

(a) Question description

Python Autocomplete

```
1  class Solution(object):
2      def longestIncreasingPath(self, matrix):
3          """
4          :type matrix: List[List[int]]
5          :rtype: int
6          """
```

(b) Question's coding environment

Time Submitted	Status	Runtime	Memory	Language
11/30/2021 22:06	Runtime Error	N/A	N/A	javascript
11/30/2021 22:06	Runtime Error	N/A	N/A	python
11/30/2021 22:06	Accepted	5 ms	39 MB	java
11/30/2021 22:05	Compile Error	N/A	N/A	c

(c) Question's submission history

Figure 2: An example LeetCode question, named Longest Increasing Path in a Matrix [16]

length of the longest increasing path in a matrix), and any assumptions (no wrap-around). Each question also comes with a coding environment to submit solutions, shown in Figure 2b. This coding environment contains the function name (`longestIncreasingPath`) and parameters (`self, matrix`) with clear details into the type of each parameter. All this information corresponds to the attributes that compose a good query context for Copilot, shown in Figure 1a.

LeetCode's coding environment also contains a set of test cases in multiple programming languages. Figure 2b shows the Python coding environment for testing a submission against LeetCode's predefined set of test cases. The availability of test cases for each query allows us to measure the behavioral correctness of Copilot's suggestions. LeetCode's tests also ensure that submitted code snippets "meet various time and space restrictions and pass corner cases" for the given problem [17]. Users are also able to see a history of their submission status for the current coding problem and any past code solutions submitted for the same question, as shown in Figure 2c. The possible statuses are:

- *Accepted*: submitted code passes all test cases
- *Wrong Answer*: submitted code has no errors, but its output is different from the expected output for at least one test case.
- *Compile Error*: submitted code cannot be compiled.
- *Time Limit Exceed*: submitted code has no errors, but at least one test case exceeds permitted execution time.
- *Runtime Error*: submitted code has at least one test case that fails due to errors during execution (i.e. division by zero, etc)

SonarQube and Understandability Metrics. Dantas et al. [9] showed that cognitive complexity and cyclomatic complexity can be used as proxy metrics for measuring the understandability of a code snippet. The authors use SonarQube [29], an open-source platform for statically analyzing code, to calculate both metrics. To calculate cyclomatic complexity, SonarQube starts with a value of 1 and increments by one whenever it detects a split in the control flow of a function [29]. Essentially, a higher cyclomatic complexity implies

more branching in the code and the need for more test cases to fully cover a method [6]. While cognitive complexity also measures understandability, it relies less on mathematical models that analyze control flow and instead uses rules that map into a programmer's intuition of how they understand code [7]. Specifically, to measure cognitive complexity, SonarQube does not increment the complexity score when shorthands are used (e.g., using a ternary expression would not increase the complexity score), increments the score only once for each break in the linear flow of the code (e.g., a whole switch statement would increment the score by 1 since it can often be taken in with one glance), and increments the score when flow-breaking structures are nested [7].

3 EMPIRICAL STUDY SETUP

We now explain how we assess Copilot's synthesized code suggestions when provided with good query context.

Step 1: Gather Prerequisites and Generate Queries: We randomly select 33 LeetCode questions with varying difficulty levels (4 easy, 17 medium, and 12 hard). Using LeetCode's data (see Section 2), we manually extract the necessary information to build good query contexts for each question: function name, parameters, input, output, and comment. We then pass this information to a custom-built script that creates a separate code file for each of Python, Java, JavaScript, and C. Overall, we create 132 code files in 4 languages.

Step 2: Acquire Copilot Suggestions: Given the lack of Copilot APIs, we perform Step 2 manually. We include the code files from Step 1 into VSCode projects, manually invoke Copilot for each query, and save the top complete function suggestion. Note that Copilot uses the provided information in the file and project to construct its own internal context, but the exact details are not publicly shared [11].

Step 3: Evaluate Correctness: For each of the 132 suggestions collected from Step 2, we manually fill LeetCode's coding environment for the corresponding question (see Section 2) with the Copilot's suggested code and submit it to LeetCode to run against its test cases. We then use LeetCode's API to automatically extract the submission results in JSON format, which include the programming language of the solution, the submission status, and the submitted code snippet. We create a Python script to analyze these JSON files and report the necessary statistics for our research questions.

Step 4: Evaluate Understandability: We run SonarQube on all 132 collected files, which now contain Copilot's solutions. Unfortunately, we were not able to run SonarQube on the C code snippets because of various configuration issues that prevented SonarQube from being able to analyze the code. Thus, we report understandability scores for only 99 solutions in Java, JavaScript, and Python.

4 EMPIRICAL RESULTS

4.1 RQ1: Are Copilot's code suggestions correct?

Results. Table 1 shows each LeetCode question, number of test cases available for that question, and how many tests the Copilot solution passes. At the bottom of the table, we show statistics of the LeetCode solution status (partially correct explained later).

Overall, in 23/33 questions (70%), shown in bold in Table 1, Copilot produces an accepted solution for at least one language. Per language, 14 (42%), 19 (57%), 9 (27%), and 13 (39%) of Copilot solutions for Python, Java, JavaScript, and C respectively are accepted

Table 1: Correctness of GitHub Copilot's suggestions for each LeetCode question and overall submission status.

	Question	# Tests	Number (%) test cases passed			
			Python	Java	JavaScript	C
Easy	**Q1**	57	57 (100%)	57 (100%)	57 (100%)	57 (100%)
	Q2	14	14 (100%)	14 (100%)	14 (100%)	14 (100%)
	Q3	34	34 (100%)	34 (100%)	34 (100%)	34 (100%)
	Q4	15	15 (100%)	15 (100%)	15 (100%)	15 (100%)
Medium	Q5	81	50 (62%)	12 (15%)	6 (7%)	50 (62%)
	Q6	596	596 (100%)	596 (100%)	0 (0%)	0 (0%)
	Q7	85	82 (96%)	85 (100%)	6(7%)	0 (0%)
	Q8	58	57 (98%)	47 (81%)	58 (100%)	58 (100%)
	Q9	116	114 (98%)	116 (100%)	0 (0%)	116 (100%)
	Q10	58	58 (100%)	58 (100%)	14 (24%)	10 (17%)
	Q11	54	54 (100%)	54 (100%)	0 (0%)	54 (100%)
	Q12	49	42 (85%)	49 (100%)	49 (100%)	24 (48%)
	Q13	47	47 (100%)	46 (98%)	11 (23%)	0 (0%)
	Q14	50	17 (34%)	50 (100%)	50 (100%)	50 (100%)
	Q15	202	202 (100%)	202 (100%)	202 (100%)	202 (100%)
	Q16	15	15 (100%)	15 (100%)	0 (0%)	14 (93%)
	Q17	188	188 (100%)	188 (100%)	36 (19%)	0 (0%)
	Q18	9	2 (22%)	0 (0%)	0 (0%)	9 (100%)
	Q19	57	0 (0%)	34 (59%)	0 (0%)	0 (0%)
	Q20	210	110 (52%)	19 (9%)	110 (52%)	0 (0%)
	Q21	15	10 (66%)	10 (66%)	0 (0%)	10 (66%)
Hard	**Q22**	84	83 (98%)	0 (0%)	84 (100%)	0 (0%)
	Q23	17	17 (100%)	17 (100%)	0 (0%)	17 (100%)
	Q24	64	64 (100%)	64 (100%)	3 (5%)	60 (93%)
	Q25	70	70 (100%)	0 (0%)	0 (0%)	0 (0%)
	Q26	30	0 (0%)	30 (100%)	0 (0%)	30 (100%)
	Q27	38	0 (0%)	38 (100%)	35 (92%)	38 (100%)
	Q28	39	1 (2%)	14 (35%)	26 (66%)	2 (5%)
	Q29	138	0 (0%)	138 (100%)	0 (0%)	0 (0%)
	Q30	51	24 (47%)	26 (50%)	0 (0%)	28 (54%)
	Q31	49	0 (0%)	39 (79%)	7 (14%)	0 (0%)
	Q32	44	0 (0%)	27 (61%)	0 (0%)	0 (0%)
	Q33	101	10 (9%)	10 (9%)	7 (6%)	5 (4%)
	Accepted		14 (42%)	19 (57%)	9 (27%)	13 (39%)
	Wrong answer		12 (36%)	12 (36%)	12 (36%)	8 (24%)
	Time limit exceeded		2 (6%)	1 (3%)	1 (3%)	1 (3%)
	Compile errors		0 (0%)	0 (0%)	0 (0%)	8 (24%)
	Runtime errors		5 (15%)	1 (3%)	11 (33%)	3 (9%)
	Partially correct		13 (39%)	11 (33%)	11 (33%)	9 (27%)

(i.e., pass *all* tests). We find that Copilot solutions rarely exceeded LeetCode's time limit and that compilation errors occur only in 8/33 (24%) of the C solutions, but in none of the Java solutions. Note that Python and JavaScript are not compiled languages so their code will naturally not result in any compilation error. With the exception of JavaScript solutions (33%), runtime errors are also not that frequent.

To collect more insights, we analyze Copilot's suggestions for Q29 (Longest Increasing Path in a Matrix) where only the Java solution is accepted. Figure 2a shows Q29's description; a simple solution is to run depth-first search (dfs) on each cell of the matrix and then return the length of the longest increasing path in the matrix. Copilot's solutions in Java, JavaScript, and Python all follow this approach and use a helper function `dfs`. However, only the Java solution contains `dfs`' implementation, causing JavaScript and Python solutions to fail at runtime. The examples provided on Copilot's homepage [11] suggest that it generates code for only

one function at a time. We contacted Copilot's team and confirmed that Copilot's suggestions should not include the implementation of helper functions. Thus, the produced Java `dfs` implementation is actually unexpected Copilot behavior. Finally, Copilot's C solution for this question contains an undefined variable compilation error, and the tests were not run. Overall, this example suggests that Copilot may generate code that relies on non-existing helper functions or that contains undefined variables, both of which would result in compile or run-time errors, depending on the language.

Implications. GitHub's internal evaluation of Copilot Python suggestions shows that Copilot achieved 43% correctness on the first try [11], which is similar to our Python results (42%). Overall, our findings suggest that Copilot's first solution is often fully correct at least 27% of the time for JavaScript, and as high as 57% of the time for Java. If we consider any solution that passed *some* test cases, regardless of final LeetCode status, as *partially correct* (bottom of Table 1), then Copilot also suggests a good starting point for an additional 27%-39% of the queries, depending on the programming language. Thus, overall, Copilot will help the developer completely solve their programming task (accepted) or provide them with a useful starting point (partially correct) in 61%-91% of the time, depending on the language. In their FAQs, GitHub mentions JavaScript, Python, and Java as three of the languages Copilot does particularly well on [11]. While C is not mentioned among these languages, we find that Copilot was still able to provide a fully correct solution or a starting point in 67% of the queries.

RQ1 Summary: Copilot's correctness (passing all tests) varies by language, with Java as highest (57%) and JavaScript lowest (27%).

4.2 RQ2: How understandable is the code provided by GitHub Copilot?

Results. Figure 3 shows the cyclomatic and cognitive complexity of Python, Java, and JavaScript Copilot solutions. We find that the median cyclomatic and cognitive complexity for the solutions in all three languages is the same (5 and 6 respectively). We also run a Wilcoxon paired signed rank test to compare the distribution of complexity values per solution and find no statistical difference between any of the languages. The only difference we observe is that JavaScript has no outlier values for cyclomatic complexity and only one for cognitive complexity, as opposed to two in the other languages. We discuss one outlier example below.

The Python solution with the highest reported cyclomatic complexity of 43 and cognitive complexity of 42 corresponds to Q14 (Integer Break). Q14's description is "*Given an integer n, break it into the sum of k positive integers, where k >= 2, and maximize the product of those integers [...]*" [15]. Interestingly, the JavaScript and Java solutions for Q14 have much lower cyclomatic and cognitive complexities, 4 and 3 respectively. In Python, Copilot produced a solution with 43 if statements for input numbers 2 to 43 and their corresponding return values. In contrast, for JavaScript and Java, there were only base cases for inputs 2 and 3, followed by a loop for all inputs greater than 4. Both JavaScript and Java solutions passed all 50 test cases, while the Python solution passed only 17/50 tests.

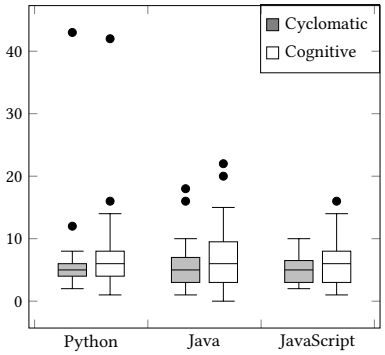

Figure 3: Cyclomatic and Cognitive Complexity Results

Implications. Copilot generally produces understandable code. The median complexity values of Copilot's solutions are well below the 15-25 thresholds typically used in static analysis tools [22]. Overall, there are only two questions where Copilot suggests code, in any of the languages, with either complexity metrics above 15. On the other side, while our data is not enough for us to draw general conclusions, the example outlier we investigated suggests that Copilot does not always generate the most compact code for a given problem. This likely depends on training data quality.

RQ2 Summary: The median cognitive complexity and cyclomatic complexity of Copilot solutions is 6 and 5, respectively, with no statistically significant differences between languages.

5 LIMITATIONS AND THREATS TO VALIDITY

Given the need to manually query Copilot, our study is limited to only 33 queries. However, we run these queries in four programming languages, resulting in 132 queries. Copilot is closed-source and we cannot map our results to the details or characteristics of Copilot's internal model. We also do not know Copilot's exact training data, which means we cannot determine if the exact solutions to our queries already exist in the data. We focus only on Copilot's functionality of converting comments to code. Our results also show Copilot's suggestions in the "best case", given our ideal query contexts. Future work is needed to study Copilot's behavior in less ideal situations and to assess its other capabilities. For reproducibility, we archive all Copilot's suggestions we received [23].

LeetCode stops execution at the first failed test case. Thus, the number of passed tests in Table 1 present a lower bound in some cases. However, this does not affect the overall status of the solution or the conclusions we draw from our study. We calculate the complexity metrics for all Copilot suggestions, including those with errors. However, this allows us to assess understandability of all suggestions a developer would see.

6 RELATED WORK

Since GitHub Copilot was just recently released, there have not been many studies into the tool. To the best of our knowledge, there are only two (publicly available) direct studies of Copilot suggestions. The first by Pearce et al. [24] examines the security issues in code generated by Copilot based on queries created from the 25 top CWE vulnerabilities with a total of 89 scenarios. In our work, we

focus on evaluating the correctness and understandability of Copilot's suggestions and do not study security issues. The second study is by Sobania et al. [27] who compare Copilot's program synthesis performance with genetic programming based on a genetic programming benchmark of 29 problems. The authors also use tests as a means to evaluate correctness. However, their study focuses only on Python and considers Copilot alternative suggestions. Our work evaluates Copilot's *first* suggestion for 33 programming problems in four different languages (132 total queries).

While there is a lot of work on code synthesis and code search using deep learning (e.g., [3–5, 8, 12, 14, 19, 32]), we do not dive into the specifics here since we focus on evaluating Copilot's capabilities while treating it as a black box. We note that the recent work by Chen et al. [8] evaluates the capabilities of Codex, a GPT language model trained on GitHub code to produce code given a natural language description. Copilot is powered by a distinct production version of Codex. In this paper, our goal is to evaluate Copilot itself as currently presented to developers.

7 CONCLUSIONS

In this paper, we presented an empirical study of the correctness and understandability of GitHub Copilot's synthesized code solutions based on 33 LeetCode questions. We evaluated Copilot's suggestions in Java, JavaScript, Python, and C. Our Python results are similar to GitHub's internal evaluation of Copilot [11]. Overall, we find that Copilot's Java suggestions have the highest correctness score (57%) while JavaScript is lowest (27%). Copilot's suggestions also have low complexity with no statistically significant differences between the complexity scores of the same solution in different languages. We also find some potential Copilot shortcomings, such as generating code that can be further simplified/compacted and code that relies on undefined helper methods.

8 ACKNOWLEDGMENTS

This research was undertaken, in part, thanks to funding from the Canada Research Chairs Program. We would also like to thank Max Schaefer for his feedback on earlier drafts of this work.

REFERENCES

[1] Romaana Aamir. 2021. GitHub copilot-bright future or an impending doom. https://code.likeagirl.io/github-copilot-bright-future-or-an-impending-doom-df0f1674a50c
[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.
[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]
[5] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.
[6] G. Ann Campbell. 2018. Cognitive Complexity: An Overview and Evaluation. In *Proceedings of the 2018 International Conference on Technical Debt* (Gothenburg, Sweden) *(TechDebt '18)*. Association for Computing Machinery, New York, NY, USA, 57–58. https://doi.org/10.1145/3194164.3194186
[7] G. Ann Campbell. 2021. Cognitive complexity - A new way of measuring understandability. https://www.sonarsource.com/docs/CognitiveComplexity.pdf
[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]
[9] Carlos Eduardo de Carvalho Dantas and Marcelo de Almeida Maia. 2021. Readability and Understandability Scores for Snippet Assessment: an Exploratory Study. *CoRR* abs/2108.09181 (2021). arXiv:2108.09181 https://arxiv.org/abs/2108.09181
[10] fabasoad and Sachin131. 2016. Is there public API endpoints available for leetcode? https://leetcode.com/discuss/general-discussion/1297705/is-there-public-api-endpoints-available-for-leetcode
[11] GitHub. 2021. GitHub Copilot · Your AI pair programmer. https://copilot.github.com/
[12] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 933–944. https://doi.org/10.1145/3180155.3180167
[13] HackerRank. [n.d.]. HackerRank for Work API. https://www.hackerrank.com/work/apidocs#
[14] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. 837–847. https://doi.org/10.1109/ICSE.2012.6227135
[15] LeetCode. [n.d.]. Integer Break. https://leetcode.com/problems/integer-break/
[16] LeetCode. [n.d.]. Longest Increasing Path in a Matrix. https://leetcode.com/problems/longest-increasing-path-in-a-matrix
[17] LeetCode. 2019. Start your coding practice –. https://support.leetcode.com/hc/en-us/articles/360012016874-Start-your-Coding-Practice
[18] LeetCode. 2021. The world's leading online programming learning platform. https://leetcode.com/
[19] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 4159–4165. https://doi.org/10.24963/ijcai.2018/578
[20] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.
[21] Matthew MacDonald. 2021. GitHub copilot: Fatally flawed or the future of software development? https://medium.com/young-coder/github-copilot-fatally-flawed-or-the-future-of-software-development-390c30afbc97
[22] Gerald Mücke and G Ann Campbell. 2021. How to use cognitive complexity? https://community.sonarsource.com/t/how-to-use-cognitive-complexity/1894/7
[23] Nhan Nguyen and Sarah Nadi. 2022. Online artifact for MSR 2022 Submission "An Empirical Evaluation of GitHub Copilot's Code Suggestions". https://doi.org/10.6084/m9.figshare.18515141
[24] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. arXiv:2108.09293 [cs.CR]
[25] Martin Robillard, Robert Walker, and Thomas Zimmermann. 2009. Recommendation systems for software engineering. *IEEE software* 27, 4 (2009), 80–86.
[26] Swapnil Rustagi and Jagga Jasoos. 2019. Access to CodeChef API. https://discuss.codechef.com/t/access-to-codechef-api/27308
[27] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2021. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of GitHub Copilot and Genetic Programming. arXiv:2111.07875 [cs.SE]
[28] SonarQube. 2021. Code quality and code security. https://www.sonarqube.org/
[29] SonarQube. 2021. Metric definitions. https://docs.sonarqube.org/latest/user-guide/metric-definitions/
[30] Meng Xia, Mingfei Sun, Huan Wei, Qing Chen, Yong Wang, Lei Shi, Huamin Qu, and Xiaojuan Ma. 2019. *PeerLens: Peer-Inspired Interactive Learning Path Planning in Online Question Pool*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300864
[31] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 2203–2214. https://doi.org/10.1145/3308558.3313632
[32] Qihao Zhu and Wenjie Zhang. 2021. Code Generation Based on Deep Learning: a Brief Review. arXiv:2106.08253 [cs.SE]