Assessing the Quality of GitHub Copilot's Code Generation

Burak Yetistiren
burakyetistiren@hotmail.com
Bilkent University
Ankara, Turkey

Isik Ozsoy
ozsoyisik@gmail.com
Bilkent University
Ankara, Turkey

Eray Tuzun
eraytuzun@cs.bilkent.edu.tr
Bilkent University
Ankara, Turkey

ABSTRACT

The introduction of GitHub's new code generation tool, GitHub Copilot, seems to be the first well-established instance of an AI pair-programmer. GitHub Copilot has access to a large number of open-source projects, enabling it to utilize more extensive code in various programming languages than other code generation tools. Although the initial and informal assessments are promising, a systematic evaluation is needed to explore the limits and benefits of GitHub Copilot. The main objective of this study is to assess the quality of generated code provided by GitHub Copilot. We also aim to evaluate the impact of the quality and variety of input parameters fed to GitHub Copilot. To achieve this aim, we created an experimental setup for evaluating the generated code in terms of validity, correctness, and efficiency. Our results suggest that GitHub Copilot was able to generate valid code with a 91.5% success rate. In terms of code correctness, out of 164 problems, 47 (28.7%) were correctly, while 84 (51.2%) were partially correctly, and 33 (20.1%) were incorrectly generated. Our empirical analysis shows that GitHub Copilot is a promising tool based on the results we obtained, however further and more comprehensive assessment is needed in the future.

CCS CONCEPTS

• **Software and its engineering → Source code generation**.

KEYWORDS

GitHub Copilot, code generation, code completion, AI pair programmer, empirical study

ACM Reference Format:
Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the Quality of GitHub Copilot's Code Generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '22), November 17, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3558489.3559072

PROMISE '22, November 17, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9860-2/22/11…$15.00
https://doi.org/10.1145/3558489.3559072

1 INTRODUCTION

GitHub Copilot[1] is a code generation tool that utilizes a variety of technologies, including a compatible IDE, and the OpenAI Codex Model[2]. GitHub Copilot can be installed and used as an extension to Visual Studio Code, Neovim, IDEs developed by JetBrains [6], and GitHub Codespaces[3]. The underlying service continuously takes code samples from the users and sends the snippets to the underlying OpenAI Codex Model. GitHub Copilot generates the code and presents the results of the OpenAI Codex Model by adjusting the generated code to the current workspace of the programmer [4]. The Codex model relies on Generative Pre-trained Transformer (GPT) models that the company previously invented for text generation. The public code available on GitHub was used during the fine-tuning of the model to implement the code recognition and generation capabilities.

There are mixed reviews about the prospect of the GitHub Copilot. On the one hand, reducing development time, easing the development process by suggesting code for small utilities, and suggesting better alternatives for code snippets are some of the positive feedback developers provided [2, 7, 13]. On the other hand, it is argued that the current state of technology is not promising enough to match human ingenuity. Considering the previous studies, the service requires a vast amount of human interaction, making the coding routine still heavily reliant on the programmer [1].

The reviews about GitHub Copilot we touched upon only include brief and heuristic feedback in terms of the evaluation of the service. We agree with the general consensus of the opinions about GitHub Copilot and find it worthwhile to evaluate the possible enhancements a service like GitHub Copilot can offer. Clearly, GitHub Copilot is capable of generating code, but its value is undetermined. To systematically evaluate GitHub Copilot, we propose to construct an experimental setup to assess the generated code in terms of validity, correctness, and efficiency. In this context, we defined the following research questions:

RQ1 What is the quality of the code generated by GitHub Copilot?
  RQ1.1 How valid are GitHub Copilot's code suggestions?
  RQ1.2 How correct are GitHub Copilot's code suggestions?
  RQ1.3 How efficient are GitHub Copilot's code suggestions?
RQ2 What is the effect of using the docstrings on the generated code quality?
RQ3 What is the effect of using appropriate function names on the generated code quality?

In the following sections, we first elaborate on our experimental setup in Section 2. In Section 3, we present the results we gathered from our setup. In Section 4, we share and evaluate our results. In

[1] copilot.github.com
[2] openai.com/blog/openai-codex/
[3] github.com/features/codespaces

Section 5, we discuss factors that might influence the validity of our results. We provide an overview of other works that study GitHub Copilot in Section 6. Lastly, we conclude our study in Section 7.

2 METHODOLOGY

In our experiment, we used HumanEval dataset [3], which is described in Section 2.1. To address the research questions, we created an experimental setup, which systematically evaluates the effectiveness of GitHub Copilot that is described in Section 2.2. The details of our assessment are presented in Sections 2.3–2.5. In Sections 2.6 and 2.7, we elaborate on the two additional experiments we conducted to test the effect of the function names and explanations of the generated code quality.

2.1 HumanEval Dataset

For our experiment, we use the HumanEval dataset [3]. This dataset contains 164 problems. Each problem is accompanied by a task ID, a prompt, the canonical solution, and unit tests. The structure of a problem can be viewed in Figure 1. The task ID is the ID of that particular problem which ranges from 0 to 163. The prompt part contains the function prototype, the explanation of the problem, some function calls and their output in a Python docstring, and library imports, if applicable. A canonical solution is a solution to the problem provided by a "human" programmer. The test part contains unit tests as a Python function.

From our literature survey[4] on GitHub Copilot, we found that the combination of code comments and function signatures yields more robust results. We pass the function prototype and the docstring as input to GitHub Copilot. An example code generation done by GitHub Copilot, where the input problem is shown in Figure 1 can be viewed in Listing 1.

```
from typing import List

def has_close_elements(numbers: List[float], threshold:
    float) -> bool:
    """ Check if in given list of numbers, are any two
    numbers closer to each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0,
    2.0], 0.3)
    True
    """
    for i in range(len(numbers)):
        for j in range(i + 1, len(numbers)):
            if abs(numbers[i] - numbers[j]) < threshold:
                return True
    return False
```

Listing 1: Generated Code for the Example Problem (ID: 0)

2.2 Experimental Setup

In Figure 2, we provide a step-by-step illustration of the experiment's workflow. Given the HumanEval problem dataset [3], we start our experiment by extracting the problems. We achieve this by reading the dataset and representing each problem contained in the dataset with a separate JSON format file. After completing the extraction procedure, we save the canonical solution, unit tests, and the prompt of a problem as separate Python files to the directory corresponding to the problem's ID. Subsequently, we generate solutions by using an already prepared Python file containing the prompt. This prompt is the combination of the function signature and docstring contained in the function body. Given the dynamic characteristic of GitHub Copilot in terms of the interactions between the programmer and the service, we implement the code generation step of our experiment manually.

After the code generation step is completed, we start the assessment phase by executing the tests on the generated solutions to assess code validity and code correctness. Afterward, we inspect the time and space complexities of the generated code and the canonical solution, by sending corresponding requests using OpenAI API[5] and compare the results. For each step of the assessment phase, we save the results of the individual assessment related to the problem. The extracted results can be seen in our reproduction package[6].

We further test the validity of our findings in our literature survey about providing GitHub Copilot with code comments and function signatures, by implementing two additional experiments about the significance of function names, parameters, and comments explained in Sections 2.6 and 2.7.

task_id

HumanEval/0

prompt

```
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

canonical_solution

```
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

test

```
METADATA = {
    'author': 'jt',
    'dataset': 'test'
}

def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False
```

Figure 1: Example Problem (ID: 0) from HumanEval dataset

[4] https://github.com/burakyetistiren/-An-Empirical-Evaluation-of-GitHub-Copilot-s-Code-Generation-/blob/main/misc/article_names_and_links.pdf

[5] openai.com/api

[6] https://github.com/burakyetistiren/-An-Empirical-Evaluation-of-GitHub-Copilot-s-Code-Generation-/blob/main/misc/Copilot_Results.pdf

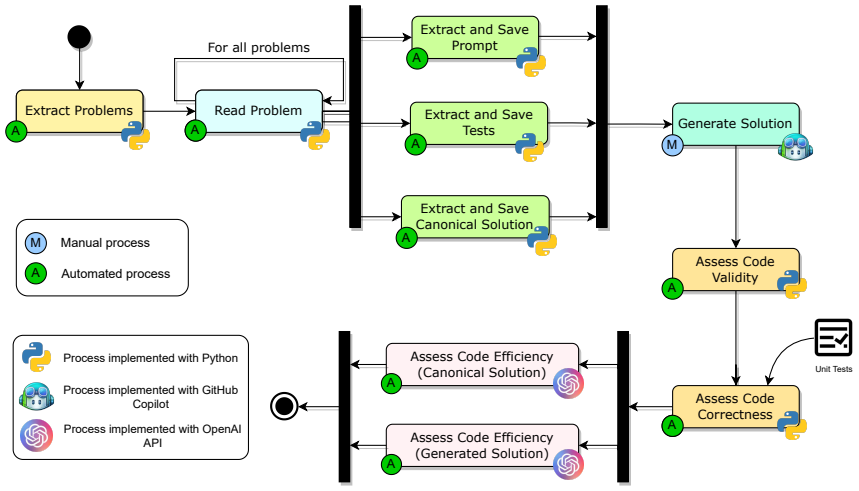

Figure 2: Experiment Workflow

2.3 Code Validity (RQ1.1)

Code validity is assessed in terms of how a given code segment is compliant with the rules and regulations (i.e., syntax rules) of a given programming language and with any errors that could be raised during runtime. The dataset we use is constructed for the Python programming language; therefore, to check for code validity, we make use of the Python 3.8 interpreter. After we initiate the program, we await any further errors that could be raised during runtime and record such cases.

2.4 Code Correctness (RQ1.2)

For code correctness, we want to assess the extent to which the generated code performs as intended. As we previously stated, the problems in the HumanEval dataset are accompanied by problem-specific unit tests. On average, each problem comes with 7.7 unit tests [3]. We measured the code correctness as passed unit tests divided by all unit tests for a specific problem.

2.5 Code Efficiency (RQ1.3)

When analyzing the given generated solution, we employ time and space complexity analysis and use the big-O: $O(f(n))$ notation, where $f(n)$ is the solution generated by GitHub Copilot, and O sets an upper bound on time and space complexities.

To obtain the time and space complexities, we use the OpenAI API. Given the canonical solution and the generated code as input, we make a query to the API, requesting the time and space complexities of both algorithms. We then evaluate GitHub Copilot's performance for that particular problem from the dataset.

2.6 Using only Function Names and Parameters Without Prompt (RQ2)

We removed the docstrings from the problems to assess the effect of docstring on the generated solution. The docstring of a given problem in the HumanEval dataset includes the explanation of the function as the intended purpose of what that problem should be doing. This explanation is then accompanied by some sample test cases and their results (an example can be seen in the "prompt" part in Figure 1). We used GitHub Copilot to generate code by only using the name and the parameters of the function as a reference. We aimed to see how our results would change in comparison to our previous results.

2.7 Using Dummy Function Names (RQ3)

We changed the function names of the problems with a dummy function name 'foo', to assess the effect of meaningful function names on the generated solution. We used GitHub Copilot to generate code using only the parameters and prompt of the function as a reference.

3 RESULTS

After we executed our pipeline in Figure 2, we saved our results in a '.csv' format file. To replicate our results, our setup could be accessed and downloaded [7] (Replication Package).

3.1 Code Validity (RQ1.1)

As we noted earlier, our metric for code validity is binary, such that if any errors were raised during the execution of a given Python script, we denoted that script as invalid.

Out of 164 generations to the problems, 14 were invalid and 150 were valid. This yielded a 91.5% success rate in terms of generating valid code.

3.2 Code Correctness (RQ1.2)

We used the number of passed unit tests divided by all unit tests to calculate the success percentage of the code. In Figure 3, we provided the percentage distribution of code generations falling under different categories (correct, partially correct, and incorrect).

We observed that for 28.7% of the problems, GitHub Copilot managed to generate the correct code for the given problem, whereas it

[7] https://github.com/burakyetistiren/-An-Empirical-Evaluation-of-GitHub-Copilot-s-Code-Generation-

completely failed to provide a correct solution for 20.1% of the problems. Generated solutions for the remaining 51.2% of the problems were partially correct. Partially correct generations are the ones that pass at least one of the unit tests but not all of them. We believe partially correct generations are useful, with the assumption that if at least one unit test is passing, this is a potential indicator that with further improvements by the programmer, the code could become correct. To analyze the partially correct code generations, we created a second pie chart in Figure 4, in which we eliminated correct and incorrect code generations, yielding 84 problems. We divided $(0, 100)$ success space into four intervals. GitHub Copilot managed a success rate of 13.1% for the interval of $100\% > N > 75\%$. Followingly, code was generated with a correctness score in the interval of $75\% \geq N > 50\%$, 28.6% of the time. The next interval contained the greatest fragment of the partially correct code generations with a score of 35.7%, belonging to the interval of $50\% \geq N > 25\%$. For the last interval of $25\% \geq N > 0\%$, the score was 22.6%.

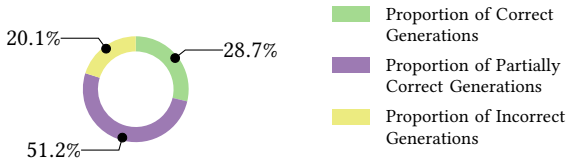

Figure 3: Distribution of Code Generations in terms of Correctness

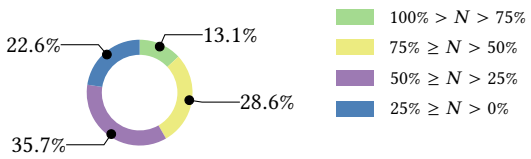

Figure 4: Distribution of Correctness Scores among Partially Correct Generations

3.3 Code Efficiency (RQ1.3)

Table 1 presents the results for time & space complexities. The rows represent functional correctness, and the columns show the efficiency comparison of the generated code and the canonical solution. The efficiency results of valid incorrect and invalid incorrect solutions are also included to eliminate any confusion that could be caused by the absence of these problems.

Time complexity: When we observed the efficiency results in terms of time complexities for the correctly generated code, for 87.2% of the problems, GitHub Copilot was as efficient as a human programmer, as the canonical solutions are hand-written in the HumanEval dataset [3]. For the remaining 12.8% of the problems, GitHub Copilot managed to generate a more efficient solution. For the partially correct code generations, the total number of problems was 84. For 64 problems out of 84, which is 76.2% of the solutions generated by GitHub Copilot yielded the same level of efficiency as the canonical solution. Whereas for 3.6% of the partially correct solutions, the code generated by GitHub Copilot was less efficient than the canonical solution, and for the remaining 20.2% of the problems, the generated solution was more efficient.

Table 1: Efficiency Results for Time Complexity (TC) and Space Complexity (SC)

	More Efficient		Same in Efficiency		Less Efficient		Total
	TC	SC	TC	SC	TC	SC	
Correct	6	2	41	42	0	3	47
Partially Correct	17	7	64	70	3	7	84
Valid Incorrect	3	5	14	12	2	2	19
Invalid Incorrect	2	1	11	13	1	0	14
Total	28	15	130	137	6	12	164

Table 2: Percentage Results for Code Correctness and Validity for the Original Experiment and the Experiment Using only Function Names and Parameters

	Original Experiment	With only Function Name
Valid	91.5%	79.3%
Correct	28.7%	19.5%
Partially Correct	51.2%	26.9%
Valid Incorrect	11.6%	32.9%
Invalid Incorrect	8.5%	20.7%

Space complexity: For the correct code generations, we had 47 problems, and 89.4% of the problems had the same efficiency as the canonical solution. On the other hand, 4.2% of the results were more, and 6.4% of the results were less efficient than the corresponding canonical solution for the given problem. For the partially correct code, we had 84 problems, where 83.4% of the problems had the same efficiency in terms of space complexities with the corresponding canonical solution. Copilot managed to generate a more efficient solution 8.3% of the time, whereas it generated a less efficient solution again 8.3% of the time.

3.4 Using only Function Names and Parameters Without Prompt (RQ2)

The results we presented up until this point were the outputs of the experiment where we provided the function name, parameters, and the docstring as the inputs to get the generated code from GitHub Copilot. In this part, as we explained in Section 2.6, we removed the docstring from each of our problems in the dataset. The results of this experiment are presented in Table 2 and Table 3.

In our original experiment where we used both the function name and the prompt, our code validity score was 91.5%. In our latter experiment, where we only used the function names, our code validity score dropped to 79.3%. For code correctness, if we compare these results to our results in this experiment, the rate of correctly generated code dropped to 19.5% (from 28.7%). The incorrectly generated code percentage increased to 53.7% (from 20.1%), and the partially correctly generated code percentage dropped to 26.8% (from 51.2%).

To give more details about the partially correctly generated code, we divided $(0, 100)$ success space into four equal intervals, similar to the assessment we explained in Section 3.2. The precise correctness results for these intervals can be observed in Table 3.

Table 3: Percentage Results for Code Correctness for Partially Correct Solutions for Given Intervals for the Original Experiment and the Experiment Using only Function Names and Parameters

	Original Experiment	With only Function Name
$100\% > N > 75\%$	13.1%	6.8%
$75\% \geq N > 50\%$	28.6%	36.4%
$50\% \geq N > 25\%$	35.7%	25.0%
$25\% \geq N > 0\%$	22.6%	31.8%

Table 4: Percentage Results for Code Correctness and Validity for the Original Experiment and the Experiment Using Dummy Function Names

	Original Experiment	With Dummy Function Name
Valid	91.5%	84.2%
Correct	28.7%	26.8%
Partially Correct	51.2%	40.2%
Valid Incorrect	11.6%	17.1%
Invalid Incorrect	8.5%	15.9%

Table 5: Percentage Results for Code Correctness for Partially Correct Solutions for Given Intervals for the Original Experiment and the Experiment Using Dummy Function Names

	With Meaningful Function Name	With Dummy Function Name
$100\% > N > 75\%$	13.1%	4.6%
$75\% \geq N > 50\%$	28.6%	40.9%
$50\% \geq N > 25\%$	35.7%	31.8%
$25\% \geq N > 0\%$	22.6%	22.7%

3.5 Using Dummy Function Names (RQ3)

In this part, as explained in Section 2.7, we prompted GitHub Copilot to generate code for the same problems, this time with dummy function names instead of meaningful, and informative function names. The results of the original and the new experiment are presented in Table 4 and Table 5. Out of 164 problems, 26 were found to be invalid and the remaining 138 solutions were valid. This yielded an 84.2% success rate in terms of generating valid code. For code correctness, we have observed that for 26.8% of the problems, GitHub Copilot generated the correct code for the given problem, whereas 51.2% of the solutions were partially correct. Generated solutions for 19.7% of the problems were incorrect.

In general, it was observed that using dummy function names instead of meaningful ones reduces the performance of GitHub Copilot. The code validity score dropped from 91.5% to 84.2%. The code correctness score among the valid solutions dropped from 48.6% to 46.8%. The proportion of correct solutions decreased from 28.7% to 26.8%, whereas the proportion of partially correct solutions decreased from 51.2% to 40.2%. Moreover, we also observed an increase in the proportion of incorrect solutions. We have observed that the proportion of valid and incorrect solutions increased from 11.6% to 17.1%, whereas the proportion of invalid and incorrect solutions increased from 8.5% to 15.9%.

4 DISCUSSION

In this section, we discuss the results of our experiment explained in Section 3.

4.1 Code Validity (RQ1.1)

As we discussed, for our 164 problems, GitHub Copilot was able to generate valid code for 150 of them, yielding a success rate of 91.5%. The results showed that all errors were generated at runtime; furthermore, there were no errors caused by code with the wrong syntax. In this regard, we can say that GitHub Copilot is competent in generating synthetically correct code. In terms of runtime errors, while they varied, the particular error we observed the most was the incorrect use of the "int()" method in Python, where the function returns the integer equivalent of the expression provided in the parentheses. The common characteristic of the problem that caused this error was to sum the digits of the numbers in an array, but GitHub Copilot failed to vouch for the '-' signs in the negative numbers and also count them as characters. We also observed additional errors in the indexing of the arrays, operations on incompatible types, and non-terminating recursions.

> GitHub Copilot is capable of generating valid code 9 out of 10 times.

4.2 Code Correctness (RQ1.2)

In contrast to a binary scenario for code correctness where the generated solution is either correct or incorrect, GitHub Copilot also generates partially correct solutions. As it can be observed in Figure 3, for 51.2% of the problems, GitHub Copilot generated partially correct code. If more than 50% code correctness score is considered a success, then we can say that GitHub Copilot is successful in generating code for 50% of the problems, instead of 28.7%. Therefore, we argue that not only the fully correct solutions should be considered a success, but the partially correct solutions should also be taken into account. This is the case because usually in regular programming practices, it is seen that the first iteration made on the written code is not correct, but over the next iterations, the code becomes correct. Therefore, we argue that GitHub Copilot is considerably successful in understanding user intent and generating the correct code. Our standpoint is further elaborated on with an example in the following paragraph.

```
def digitSum(s):
    """Task: Write a function that takes a string as
     input and returns the sum of the upper characters
     only' ASCII codes.
    Examples:
        digitSum("") => 0
        digitSum("abAB") => 131
        digitSum("abcCd") => 67
        digitSum("helloE") => 69
        digitSum("woArBld") => 131
        digitSum("aAaaaXa") => 153
    """
    sum = 0
    for c in s:
        if c.isupper():
            sum += ord(c) - ord('A') + 1
    return sum
```

Listing 2: Problem (ID: 66) with Low Success Rate

When we analyzed why the generated code was incorrect, or partially correct with a poor success rate, we saw that with slight possible adjustments that could be made on top of the generated code, the correctness of the solution could be easily improved. The correctness of the solution to the given problem shown in Listing 2 is 13%, which falls into the lowest category shown in blue in Figure 4. But by only removing the subtraction of *"ord('A') + 1"* from *"ord(c)"* under the 'if' statement, in other words changing the line to be *"sum += ord(c)"*, we can improve the correctness of this function by 87%, to be 100%. Hence, the code generations having low correctness scores should not be considered complete failures, as we have seen that even with small adjustments the code could be fixed.

```
def get_positive(l: list):
    """Return only positive numbers in the list.
    >>> get_positive([-1, 2, -4, 5, 6])
    [2, 5, 6]
    >>> get_positive([5, 3, -5, 2, -3, 3, 9, 0, 123, 1,
     -10])
    [5, 3, 2, 3, 9, 123, 1]
    """
    return [x for x in l if x > 0]

def odd_count(lst):
    """Given a list of strings, where each string
     consists of only digits, return a list. Each element
      i of the output should be "the number of odd
     elements in the string i of the input." where all
     the i's should be replaced by the number of odd
     digits in the i'th string of the input.
    >>> odd_count(['1234567'])
    ["the number of odd elements 4n the str4ng 4 of the 4
     nput."]
    >>> odd_count(['3',"11111111"])
    ["the number of odd elements 1n the str1ng 1 of the 1
     nput.",
     "the number of odd elements 8n the str8ng 8 of the 8
     nput."]
    """
    return [str(sum(1 for c in s if c in '13579')) for s
     in lst]
```

Listing 3: Generated Solutions for a Simple (ID: 30) and More Complicated Problem (ID: 113)

We also observed that GitHub Copilot was able to solve some of the problems (e.g., arithmetic operations, searching in an array, and other basic operations) more successfully, whereas it struggled to generate a correct solution for other problems. Two such samples can be examined in Listing 3. In the first problem, it is asked to filter only the positive numbers in a given array, which is a simple and prevalent problem. In the second one, we have a more complicated problem to solve, where first, the number of odd numbers in a given string would be counted, then the 'i' characters in the string would be replaced with the found number. In the samples where the solution was not able to satisfy any of the test cases for the problem, we observe that GitHub Copilot should be provided with additional information. Moreover, as the problem consists of small subproblems which are finding the number of odd numbers in the string, and replacing 'i' characters with the numbers, adopting a step-by-step approach would be more feasible for such a problem. For example, in Listing 4, we had a task to return a tuple containing the number of odd and even palindromes in a given range. This is similar to the complex problem we stated, in terms of containing subproblems and their difficulty. Different from the previous one, two additional functions to solve the subproblems for checking if a number is an integer, and checking if a number is a palindrome are generated. This is a special case, as, in the main function, the generated code called two other functions, which were not generated yet. Subsequently, GitHub Copilot generated these two functions one after another, as we put new lines for GitHub Copilot to proceed with code generation. This special case allows us to exemplify the step-by-step approach we discussed. We show that if the implicit subproblems are solved separately, and step-by-step, the success rate in code correctness could be improved, as the success rate of this problem is 100%, whereas the solution to the previous problem is incorrect. Therefore, we can say that for complex problems, more information (i.e., code snippets, comments) should be given to GitHub Copilot. Then, the problem should be divided into simpler subproblems, and the solutions to those subproblems should be generated one by one.

> Our results show that without the supervision of a programmer, GitHub Copilot has a lower chance of generating correct code; but with further input provided by the programmer, the correctness of the generated code can be increased.

```
def even_odd_palindrome(n):
    """Given a positive integer n, return a tuple that
     has the number of even and odd integer palindromes
     that fall within the range(1, n), inclusive.
    Example 1:
        Input: 3
        Output: (1, 2)
        Explanation:
        Integer palindrome are 1, 2, 3. one of them is
     even, and two of them are odd.
    Example 2:
        Input: 12
        Output: (4, 6)
        Explanation:
        Integer palindrome are 1, 2, 3, 4, 5, 6, 7, 8, 9,
      11. four of them are even, and 6 of them are odd.
    Note:
        1. 1 <= n <= 10^3
        2. returned tuple has the number of even and odd
     integer palindromes respectively.
    """
    return (sum(1 for i in range(1, n+1) if is_palindrome
     (i) and is_even(i)),
            sum(1 for i in range(1, n+1) if is_palindrome
     (i) and not is_even(i)))

def is_palindrome(n):
    """
    Given a positive integer n, return true if it is a
     palindrome, and false otherwise.
    """
    return str(n) == str(n)[::-1]

def is_even(n):
    """
    Given a positive integer n, return true if it is even
     , and false otherwise.
    """
    return n % 2 == 0
```

Listing 4: A Special Case of Code Generation (ID: 107)

4.3 Code Efficiency (RQ1.3)

For time and space complexities and different code correctness categories (correct, partially correct, incorrect, and invalid), the generated code was mostly as efficient as the canonical solution. This could be considered a success of GitHub Copilot. Moreover, there are problems where the generated solution was more efficient than the canonical solution, which is interesting and worthy of consideration. For instance, in Listing 5, the first solution in the upper box is the canonical solution, whereas the one in the lower box is generated by GitHub Copilot. The problem asks to create a list containing all prime numbers until a given number. In the canonical solution, the approach is to iterate up until the given number 'n', and at each number 'i', iterate until 'i' and check if any number less than 'i' divides 'i'. In the generated solution, instead of checking every number less than 'i', the approach of checking up until the square root of 'n' is used. GitHub Copilot also generated the function called 'is_prime' to be used in the solution. This way, a more efficient algorithm for solving the same problem was implemented.

```
def count_up_to(n):
    primes = []
    for i in range(2, n):
        is_prime = True
        for j in range(2, i):
            if i % j == 0:
                is_prime = False
                break
        if is_prime:
            primes.append(i)
    return primes
------------------------------------------------------------
def count_up_to(n):
    """Implement a function that takes an non-negative
     integer and returns an array of the first n integers
      that are prime numbers and less than n.
    for example:
    count_up_to(5) => [2,3]
    count_up_to(11) => [2,3,5,7]
    count_up_to(0) => []
    count_up_to(20) => [2,3,5,7,11,13,17,19]
    count_up_to(1) => []
    count_up_to(18) => [2,3,5,7,11,13,17]
    """
    return [i for i in range(2, n) if is_prime(i)]

def is_prime(n):
    """Write a function that returns true if the given
     number is a prime number and false otherwise.
    Example:
    is_prime(7) == True
    is_prime(8) == False
    is_prime(2) == True
    is_prime(1) == False
    is_prime(0) == False
    """
    import math
    for i in range(2, int(math.sqrt(n))+1):
        if n % i == 0:
            return False
    return True
```

Listing 5: Comparison of the Canonical and the Generated Solutions for a Problem (ID: 96) in HumanEval Dataset

Even though there are cases where the generated code was more efficient, like the example in Listing 5, we cannot state that GitHub Copilot is necessarily successful in generating more efficient code than the canonical solutions. As we previously stated, according to our results, the bigger proportion of time and space complexities for any category of code correctness was indicating that the efficiencies of the generated and canonical solutions were the same. Furthermore, similar to the problems where the generated code is more efficient than the canonical solution, the inverse case where the canonical solution overpowers the generated code has a small fraction of the total problems, which can be seen in Table 1.

> Our results show that there is no significant difference in terms of code efficiency between the code generated by GitHub Copilot and a human programmer.

4.4 Using only Function Names and Parameters Without Prompt (RQ2)

According to the results in Section 3.4, we observed a significant drop in success for both the code correctness and code validity metrics. For example, the code validity dropped by 12.2% after the docstrings were removed from the problems, similarly, the code correctness score for correct generations dropped by 9.1%. These results reflected a general performance drop affecting the validity and the correctness of the code. From this, we argue that the code generation performance of GitHub Copilot is correlated with the explanation given as input for code generation.

There were some cases, where we did not see any decrease in correctness or validity scores. Such problems constituted 12.2% of the dataset for code correctness and 3.7% for code validity. We have seen such cases mostly for problems that include substring search, value manipulations in an array, and character comparison. Additionally, the names of such functions, accompanied by parameter names were self-explanatory, which means that GitHub Copilot could still make interpretations about the function without requiring more details.

For the cases where the code correctness and validity scores dropped, we observed that these problems were more complicated. When we examined where the success rate of GitHub Copilot dropped, we observed cases where the function name and the parameters alone failed to give details. This means that the name and parameters alone, are not sufficient to give details about such functions. For example, in one case we observed a function called "will_it_fly", which only has two parameters called 'q' and 'w'. Copilot could not understand the purpose of the function and compared the two parameters if one of them was greater than the other. The purpose of the function was to check if 'q' was a palindromic list and if the sum of the elements in the list was less than 'w'. In this problem, per the syntax of Python, Copilot cannot know the variable types of the parameters.

> While employing GitHub Copilot, the utilization of proper explanations of the given problem is important in terms of acquiring correct and valid code. If possible the programmer should provide an explanation of the problem accompanied by some sample unit tests as a docstring, comment, etc. while generating a solution.

4.5 Using Dummy Function Names (RQ3)

The aim of using dummy function names is to assess the effect of the function name on the generated code. We observed different results during our experiment. Five solutions were invalid in the experiment with meaningful function names but valid in the experiment with dummy function names. For example, problem #108 describes a function that takes an array of integers and returns the number of elements whose sum of digits is positive. In this problem, the function name can be considered misleading, therefore, using a dummy function name may eliminate the ambiguity, so GitHub Copilot was able to generate a valid solution. Similarly, in problem #99, the function name was given as "closest_integer", which can be misleading in terms of the parameter type. This is because, when we used the function name, GitHub Copilot considered the input type as float instead of a string. When we used a dummy name, it generated a solution that takes a string as explained in the function prompt.

Another observation is the change in code correctness. Three solutions were correct in the experiment with meaningful function names but incorrect in the experiment with dummy function names. For example, in problem #79, the function name can be considered informative, given as "decimal_to_binary". Therefore, when we changed the function name to "foo", GitHub Copilot generated a solution with less information as input and this resulted in an incorrect solution. We can make this statement for problems #14 and #34 as well.

It can be concluded that changing the meaningful function names with dummy function names reduced the performance of Copilot for most of the problems. The exceptional problems were the ones in which either the function name is informative or misleading. Considering the average results given in Tables 4 and 5, it can be stated that generally changing meaningful function names to dummy function names affects the performance of GitHub Copilot negatively.

> Choosing a meaningful name for a given function positively affects GitHub Copilot's performance in terms of generating a correct and valid code.

In Figure 5, we demonstrate the code correctness results using a violin chart. We show the mean value with a red and the median value with a green dashed line. For our three experiments which are the original experiment, experiment with only function names, and experiment with dummy function names, the median values are found to be 0.53, 0, and 0.44 respectively. The mean values are found to be 0.54, 0.31, and 0.46 respectively. It can be observed that we obtain better code correctness results when we use the full prompt, more information, as in the original experiment. Furthermore, it could be observed that the utilization of the docstring for explaining the intent of a given function is more essential than giving it a meaningful name. Hence, in practical uses, programmers should prioritize giving proper explanations to the functions to gather correct code. However, giving functions meaningful names is still essential per our results, therefore a combination of a proper explanation and a meaningful function name is the ideal case.

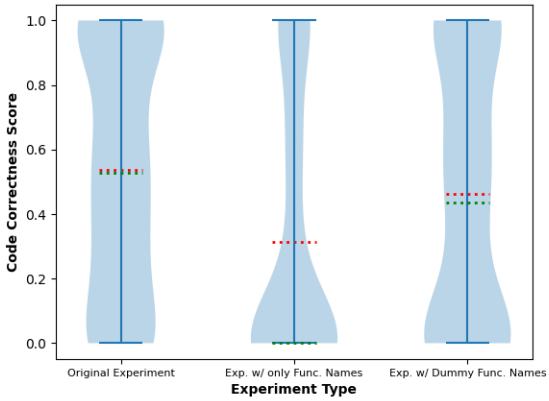

Figure 5: Code Correctness Score Distribution of the Problems for Different Experiments

5 THREATS TO VALIDITY

In this section, we discuss the possible factors of our experimental setup and GitHub Copilot that may reduce the validity of our findings.

5.1 Conclusion Validity

OpenAI Codex Model Version: While conducting our experiment, the latest version of OpenAI Codex Model was *code-davinci-001*. For any possible later evaluations of code efficiency with the same experimental setup, the results might be different.

Trivial Solutions: In some problems, GitHub Copilot generated solutions to return simple statements like empty arrays or Boolean values. In this case, if there are test cases related to the problem, where such expressions are the desired output, those test cases pass by chance without any algorithm generated for the problem.

5.2 Internal Validity

One-shot code generation: While generating code with GitHub Copilot, we used the function names, parameters, and the corresponding docstring containing an explanation of the function and a few instances of tests for that function. Furthermore, we did not write any code to provide additional information to GitHub Copilot which would clarify more what our intent for that particular problem is. Therefore, for most cases, the success rate could be increased if we have given hints as code snippets to GitHub Copilot.

Reproduction of the Generations: While conducting our experiments, we observed that GitHub Copilot had a nondeterministic characteristic, hence was generating different outputs for the same input for different times we generated code in our trials prior to our experiment. We paid great attention not to including different outputs for the same input by generating code for our problems in one iteration, and saving the generated code, then conducting our evaluation on the saved code. Given that GitHub Copilot has a dynamic characteristic that the underlying AI model of the tool

is being retrained as new repositories are added to GitHub, our results might not be fully replicated given our experimental setup and input.

Code Generation Methods: With GitHub Copilot, one can use two different approaches to generate code. The first one happens automatically as a programmer proceeds to write code, GitHub Copilot suggests code snippets that might fit into that context. In the other approach, whenever the programmer wants to generate code, they press the 'ctrl + enter' key combination to see up to 10 code generations GitHub Copilot produces. In our experiment, we chose the first approach whenever possible, otherwise, we implemented the second approach and selected the suggestion at the top of the list. For 15 problems, GitHub Copilot failed to generate any code after we entered the next line (after the user presses the 'enter' key and continues from the next line). Therefore, we had to apply the 'ctrl + enter' key combination to see the solutions, and we were able to obtain code generations for all problems. As we had to use two different methods for code generation, we stated our practice as a possible factor to reduce the validity of our study.

On a further note, we also want to state the difference between the solutions that are automatically generated, and the ones have shown when the 'ctrl + enter' key combination is applied. We observed that for the same context, two methods yield different results. Therefore, if in both methods, the code is generated, choosing different methods for a set of problems may introduce possible invalidity to a study. Hence, we tried to be as consistent as possible in our experiment by avoiding the latter method whenever possible.

Block and Line-by-Line Generation: We stated that for most of the cases GitHub Copilot managed to generate the solution of a given function as bulk, but there were cases, where we had to generate the solution line-by-line. As we had no control over how GitHub Copilot would generate the code, we had to accept the method GitHub Copilot would choose for a particular problem. We state these cases, as in line-by-line suggestion, the previously generated lines might have an effect on the next line to be generated, whereas in the first case code is generated at once as a bulk.

GitHub Copilot Version: While conducting our experiment, the latest version of GitHub Copilot was *v1.7.4421*. For any possible later evaluations with the same experimental setup, the results might be different.

5.3 Construct Validity

OpenAI API: We relied on the correctness of OpenAI to evaluate time and space complexities. Therefore there may be instances where the code efficiency results are incorrect.

Number of test cases: The varying amount of test cases for the dataset may introduce a threat to our experiment. On average there are 7.7 test cases for each problem in the HumanEval dataset [3]. Having broader test cases, both for the amount and the scope can be important. By extending the test cases, any potential corner case that could be missed may be covered. This can be critical especially when some corner cases for a given problem are not involved. We plan to improve the test cases both in quantity and quality in our future work.

5.4 External Validity

Problem Coverage: For our experiment, we evaluated the generated solutions for 164 different problems, contained in the HumanEval dataset. In the HumanEval dataset, the subjects of the problems include algorithms, simple mathematics, reasoning, and language comprehension [3]. For better and more insightful results, the number of problems can be increased, and the comprehension of the problems could be broader. For instance, in the experimental setup proposed by Xu et al. [15] for their code generation and retrieval tool, the scope of the problems consists of basic Python, file, OS, web scraping, web server & client, data analysis & ML, and data visualization. Such topics could be included in our dataset to both broaden the comprehension and increase the number of our problems. We consider this task as future work for our study.

6 RELATED WORK

In the last few years, code generation has attracted attention from researchers such as [5, 8, 12, 16]. In this study, we focus on GitHub Copilot which seems to be the first well-established instance of an AI pair-programmer.

The underlying model of GitHub Copilot, Codex, is externally developed by OpenAI and employed by GitHub. Some of the earlier versions of the current Codex model used by GitHub Copilot were evaluated by Chen et al. [3]. The Codex model relies on GPT models that OpenAI previously developed for natural language generation. The public code available on GitHub was used here while fine-tuning the model to implement the code recognition and generation capabilities. Furthermore, the model can recognize some other elements such as function signatures, code comments, etc. The model can use such elements as inputs and generate related outputs. They found that a success rate of 70.2% could be reached in terms of code correctness, by generating 100 solutions for each problem and choosing the most successful one among them. The success rate was found to be only 28.8% for the case with one solution per problem, which is consistent with our results. In this study, we provided a detailed version of this assessment. We evaluated the time and space complexities for generated solutions. Moreover, in order to extend the coverage of our study, we adjusted the HumanEval dataset by changing meaningful function names with the dummy name "foo" and regenerated the solutions.

There are also experiment-based studies similar to ours, conducted to evaluate GitHub Copilot. However, since GitHub Copilot is a considerably new tool, there are not many studies directly related to it. We list the available studies in the following.

One such study is conducted by Sobania et al. [11] in which the code correctness of GitHub Copilot is evaluated, and the tool is contrasted to the automatic program generators having the Genetic Programming (GP) architecture. They found that there is not a significant difference between the two approaches on the benchmark problems; however, the program synthesis approaches are not sufficient in supporting programmers compared to GitHub Copilot.

An evaluation of GitHub Copilot in terms of the security of the generated programs was implemented by Pearce et al. [10]. They evaluated the vulnerabilities in the generated code by Copilot. It was determined that 40% of generated programs were vulnerable.

Another study discusses the effects of GitHub Copilot by conducting a within-subjects user study [14]. It was found that GitHub Copilot did not cause a significant improvement in terms of speed and success rate. However, it was stated that most participants preferred to use Copilot in daily programming tasks since it saved the effort for the basic tasks.

Nguyen et al. [9] evaluated GitHub Copilot using 33 different LeetCode questions and four different programming languages (Python, Java, JavaScript, and C). Their evaluation includes code correctness and code understandability for the generated code. They evaluated code correctness by measuring the ratio of passed tests for each question, which is a similar approach to our study. Code understandability was measured by two different metrics, which are cognitive and cyclomatic complexity. In terms of code correctness, Java had the highest (57%) and JavaScript was the lowest (27%) score. For code understandability, they determined that there was no statistical significance between the programming languages.

In general, most of the related works evaluated the quality of code generated by GitHub Copilot or OpenAI's Codex model. The majority of the studies focused on the evaluation of code correctness, with the exception of the studies of Pearce et al. [10] where the focus is code security, and Vaithilingam et al. [14] as their work mostly concentrates on the practical usage performance of GitHub Copilot. To the best of our knowledge, this is the first study that evaluates GitHub Copilot in terms of code correctness, code validity, and code efficiency. In this sense, we believe that our methodology and results will contribute to the ongoing research about the capabilities of GitHub Copilot and other code generation tools.

7 CONCLUSION

In our study, we performed an analysis on GitHub Copilot to assess the quality of code generation from correctness, validity, and efficiency perspectives. We found that GitHub Copilot was able to generate valid code for 91.5% of the problems in the HumanEval problem dataset. 28.7% of solutions were correct, 51.2% were partially correct, and 20.1% of them were incorrect. In terms of efficiency, we observed that Copilot could mostly match the efficiency of solutions written by humans.

To understand and evaluate the impact of input parameters' quality on GitHub Copilot, we first assessed the effect of providing only function names and parameters. Based on the results, 79.3% of the solutions were valid compared to the 91.5% validity rate of the original setup. For code correctness, we obtained a success rate of 19.5% for the correct, 26.8% for the partially correct, and 53.7% for the incorrect code. We then assessed the impact of providing dummy function names and observed a drop to 84.2% in the validity rate. In terms of correctness, we found that 26.8% of the generations were correct, whereas 51.2% of them were partially correct.

Overall, the results suggest that GitHub Copilot is a promising tool. In the near future, AI pair programming tools like GitHub Copilot would potentially have a high impact on how we develop software.

As future work, we aim to both increase the number of problems and diversify the coverage and the difficulty level of problem scenarios. With minor modifications and customizations, the underlying experimental framework that we proposed could be reused to evaluate other code generation algorithms in the future. To increase the precision of the results, we plan to increase the number of unit tests for each problem. Finally, to extend our results and discussion, we plan to assess the code quality of GitHub Copilot using maintainability and reliability metrics.

REFERENCES

[1] Matt Asay. 2021. GitHub copilot isn't changing the future. https://www.infoworld.com/article/3625517/github-copilot-isnt-changing-the-future.html

[2] Scott Carey. 2021. Developers react to github copilot. https://www.infoworld.com/article/3624688/developers-react-to-github-copilot.html

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

[4] Neil A. Ernst and Gabriele Bavota. 2022. AI-Driven Development Is Here: Should You Worry? *IEEE Software* 39, 2 (2022), 106–110. https://doi.org/10.1109/MS.2021.3133805

[5] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. https://doi.org/10.48550/ARXIV.1808.10025

[6] JetBrains. 2022. *GitHub copilot - intellij IDES plugin: Marketplace.* Retrieved April 25, 2022 from https://plugins.jetbrains.com/plugin/17718-github-copilot

[7] Renato Losio. 2021. GitHub previews copilot, an openai-powered coding assistant. https://www.infoq.com/news/2021/07/github-copilot-pair-programmming/

[8] Chen Lyu, Ruyun Wang, Hongyu Zhang, Hanwen Zhang, and Songlin Hu. 2021. Embedding API dependency graph for neural code generation. *Empirical Software Engineering* 26, 4 (21 Apr 2021), 61. https://doi.org/10.1007/s10664-021-09968-2

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

[10] 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. https://doi.org/10.48550/ARXIV.2108.09293

[11] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of Github Copilot and Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 1019–1027. https://doi.org/10.1145/3512290.3528700

[12] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A Tree-Based Transformer Architecture for Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 05 (Apr. 2020), 8984–8991. https://doi.org/10.1609/aaai.v34i05.6430

[13] Darryl K. Taft. 2021. GitHub copilot: A powerful, controversial autocomplete for developers. https://thenewstack.io/github-copilot-a-powerful-controversial-autocomplete-for-developers/

[14] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[15] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 29 (mar 2022), 47 pages. https://doi.org/10.1145/3487569

[16] Maosheng Zhong, Gen Liu, Hongwei Li, Jiangling Kuang, Jinshan Zeng, and Mingwen Wang. 2022. CodeGen-Test: An Automatic Code Generation Model Integrating Program Test Information. https://doi.org/10.48550/ARXIV.2202.07612