Using GitHub Copilot for Test Generation in Python: An Empirical Study

Khalid El Haji
khalid.el.haji@gmail.com
Delft University of Technology
The Netherlands

Carolin Brandt
c.e.brandt@tudelft.nl
Delft University of Technology
The Netherlands

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
The Netherlands

ABSTRACT

Writing unit tests is a crucial task in software development, but it is also recognized as a time-consuming and tedious task. As such, numerous test generation approaches have been proposed and investigated. However, most of these test generation tools produce tests that are typically difficult to understand. Recently, Large Language Models (LLMs) have shown promising results in generating source code and supporting software engineering tasks. As such, we investigate the usability of tests generated by GitHub Copilot, a proprietary closed-source code generation tool that uses an LLM. We evaluate GitHub Copilot's test generation abilities both within and without an existing test suite, and we study the impact of different code commenting strategies on test generations.

Our investigation evaluates the usability of 290 tests generated by GitHub Copilot for 53 sampled tests from open source projects. Our findings highlight that *within* an existing test suite, approximately 45.28% of the tests generated by Copilot are passing tests; 54.72% of generated tests are failing, broken, or empty tests. Furthermore, if we generate tests using Copilot *without* an existing test suite in place, we observe that 92.45% of the tests are failing, broken, or empty tests. Additionally, we study how test method comments influence the usability of test generations.

ACM Reference Format:
Khalid El Haji, Carolin Brandt, and Andy Zaidman. 2024. Using GitHub Copilot for Test Generation in Python: An Empirical Study. In *5th ACM/IEEE International Conference on Automation of Software Test (AST 2024) (AST '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644032.3644443

1 INTRODUCTION

The act of writing unit tests is a critical but tedious task for software engineers, considered one of the most "labor-intensive tasks in software testing" [3]. In part due to the time-consuming and repetitive act of writing unit tests, software engineers frequently neglect writing unit tests [7][11]. Automatic unit test generation has been one approach to address the labor-intensive aspect of writing unit tests. Tools such as EvoSuite and Randoop are two well-known tools for automatically generating tests [13, 22]. These automatic unit test generation tools respectively rely on search-based optimization and feedback-directed techniques for test generation. While tests generated with these approaches achieve high structural coverage [24], the readability and understandability of the tests are worrisome [10, 14, 15]. Furthermore, the application of these automatic unit test generation tools in the industry is limited, in part due to software engineers having to spend a considerable amount of time analyzing the output of such tools when using them [4, 14].

GitHub Copilot[1] is a commercial code generation tool that uses a LLM to produce code suggestions (henceforth, called generations) based on comments and code context. LLMs are generative language models that are built using deep learning techniques (often employing a Transformer-based [26] architecture. Copilot uses a version of the OpenAI's[2] Codex LLM. Copilot's generations have been shown to improve perceived developer productivity in a study from GitHub itself [30]. LLMs demonstrate promising results in numerous software engineering tasks, such as programming language translation [27], code completion [12], and code summarization [1]. LLMs are capable of "producing natural-looking completions for both natural language and source code" [23]. Fittingly, research on Copilot's generations has shown that Copilot produces readable and understandable generations, with similar complexity to human-written code [2, 20].

We hypothesize that GitHub Copilot can make writing tests a less time-consuming and tedious act for software engineers. However, a crucial element to investigate is how *usable* the generated tests are, i.e., Copilot might be able to generate a unit test, but that does not mean that the test can be (directly) used by a software engineer as the test may contain a syntax or runtime error. To evaluate the usability of Copilot's test generating ability we define several aspects of usability. In general, we define a usable generation as a generation from Copilot that could be directly used in a test suite without any modification. We consider usability to be a range and not binary. In particular, we consider the following aspects for usability:[3] **Syntactic Correctness**, **Runtime Correctness**, **Passing** (the generated test should be a non-empty passing test), and **Coverage** (the generated test should cover the same branches as the same test written by a human).

A test can be generated within and without the context of an existing test suite. Copilot uses code context (code and comments) to produce its generations. Thus, test generation made within an existing test suite (test code context) may be influenced by the surrounding code context. Hence, we investigate the usability of tests generated within an existing test suite (RQ_1) and without an

AST 2024, April 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0588-5/24/04…$15.00
https://doi.org/10.1145/3644032.3644443

[1] GitHub Copilot: https://github.com/features/copilot, last visited October 23rd, 2023.
[2] OpenAI: https://openai.com/, last visited October 23rd, 2023
[3] These aspects are partially based on earlier work by Schäfer et al. [23] and Xie et al. [28] who devised ways to evaluate the "quality" of LLM-generated tests.

Project	Domain	Provider	LOC	# Test Classes	# Tests Methods
click	CLI	GitHub	21371	0	327
pyexperiment	research	GitHub	6492	36	239
django-multiurl	web development	GitHub	266	1	8
python-crontab	DevOps	GitLab	1927	19	176
exif	image handling	GitLab	6007	7	51
python-lottie	file manipulation	GitLab	24307	35	195
pyspread	GUI application	GitLab	21481	14	173

Table 1: List of open-source Python projects from which tests were sampled.

existing test suite (RQ_2). The latter scenario is especially important to understand the usability of generated tests when no tests have been written yet. By varying the code and code comments a software engineer writes in their code file before invoking Copilot, they can influence the generation they receive from Copilot. This begs the question of how code comments should be formulated to attain the most usable test generation. In particular, we evaluate the usability of test generations using different test method comment strategies both within (RQ_3) and without (RQ_4) an existing test suite. In summary, we intend to address the following questions:

- **RQ$_1$** How usable is a test generated by GitHub Copilot *within* the context of an existing test suite?
- **RQ$_2$** How usable is a test generated by GitHub Copilot *without* the context of an existing test suite?
- **RQ$_3$** How should a test method comment be formulated to attain a usable test generation from GitHub Copilot *within* the context of an existing test suite?
- **RQ$_4$** How should a test method comment be formulated to attain a usable test generation from GitHub Copilot *without* the context of an existing test suite?

2 STUDY DESIGN

In order to study the usability of tests generated we randomly sampled tests from open-source projects and then invoke Copilot to "regenerate" the tests in different scenarios (each research question is a different scenario).

2.1 Project Selection

We select seven open-source Python projects for our evaluation. We chose Python projects as Python is the best supported programming languages for Codex [8]. We now describe how we selected the projects, sampled tests, and defined our aspects of usability.

Projects were selected from GitLab[4] and GitHub. Codex has been trained on public GitHub open-source projects [8]. Caution must be exercised on selecting projects to prevent Copilot from simply regurgitating training data from GitHub. Hence, we include open-source projects from GitLab. We have intentionally selected mostly less popular projects, as we conjecture that Codex has been trained on source code from more popular open-source projects. Similar LLMs found in the literature often use popular open-source projects for their pre-training [12, 27]. Although, we ultimately can only speculate which projects Codex has been trained on. Furthermore, we only consider projects for which the test suites use the pytest[5] or unittest[6] framework to simplify the coverage analysis we need. The final set of projects in Table 1 were selected to have a diverse set of software domains ranging from a GUI application to simple file manipulation, and to have a diverse range of project sizes.

2.2 Sampling and Manual Labelling

Early manual evaluation of a subset of test methods indicated that the presence of a method comment influences the test generated by Copilot. Hence, we split the test methods into two strata: test methods with comment and test methods without comment. We then employ stratified sampling to select tests. We randomly select one test method with a comment and one without a comment for each project. This results in a batch (set of tests) containing sampled tests with and without comments for all projects. Some projects have no or only a few tests with comments, and vice versa. Hence, the batch size can vary each time a new one is created.

For each sampled test T_i we use Copilot to create a generation G_i. We call T_i the original, or human-written test. Every pair (T_i, G_i) is assigned code aspect labels. Code aspect labels intend to reveal the deficiencies of generations (such as the runtime or syntax errors). These labels are iteratively created based on manual inspection of a generation G_i and original test T_i. For example, a generated test might be failing due to not catching an exception. This pair would then be assigned a label such as `failure_to_catch_exception` among other code aspect labels. We continue to create batches of test methods until we reach a point of theoretical saturation for the code aspect labels. This initial set of test pairs (T_i, G_i) resulting from the sampling until saturation is called O.

2.3 Aspects of Usability

Recall that we define a usable generation as a test generation that could be directly used in a test suite without any modification. We define and justify our usability aspects as follows:

Syntactic Correctness. Syntax errors occur when a generation G_i can not be parsed by Python due to incorrect syntax usage. In turn, this renders the generation G_i as broken; as a generation with a syntax error requires modification before it can be employed in a test suite. Hence, negatively impacting the usability.

Runtime Correctness. A generation without syntax errors may still contain runtime errors. For example, this occurs when a generation G_i is passing an incorrect value to a parameter of a method. Similar to a generation with a syntax error, a generation with a runtime error requires modification and thus negatively impacts usability.

Passing. A syntactically correct and runtime error-free generation may still be a failing test. We prefer a passing test generation over a failing one, as it requires fewer modifications to use in a test suite. Nevertheless, a failing test generation can expose new faults. It is possible that the oracles in the generated test are correct, but the code under test (CUT) is flawed. Similarly, a passing test might unintentionally comply with faulty behavior exhibited by the CUT. We assume that the CUT is correct, because of the original test T_i passing. Hence, a generation that is a failing test requires modification before it can be employed in a test suite, which negatively impacts the usability.

[4]GitLab: https://about.gitlab.com/, last visited October 25th, 2023.
[5]pytest: https://docs.pytest.org/en/7.2.x/
[6]unittest: https://docs.python.org/3/library/unittest.html

Coverage. When considering a test pair (T_i, G_i) we can determine the branches covered by the two tests T_i and G_i separately. When the same branches are covered by G_i as in T_i, we consider the generation G_i as a more usable generation than when G_i covers fewer of the same branches. If G_i covers less branches than T_i, G_i would require modification to ensure it covers all branches as in the "intended" human-written test T_i. This intention is visible to Copilot because we leave the original method signature (and method comment). We consider covering fewer of the same branches as the original test, to negatively impact the usability of the test.

Syntactic Correctness, Runtime Correctness, and Passing are determined for all Copilot generations using the iteratively assigned code aspects labels.

2.4 Invoking Generations

We invoke a generation from Copilot for a given original test T_i by stripping the test method body, and then "regenerating" the test method body using Copilot directly in an IDE.[7] For example, in Listing 1 we have a test method from the `pyspread` project. In Listing 2 we have the same test method but stripped. This stripped test method would then be used to invoke a generation from Copilot. The token `[INSERT]` indicates from which position a Copilot generation is requested, this token is not included in the final stripped test used for invoking a generation.

```
def test_set_row_height(self):
    """Unit test for set_row_height"""

    self.data_array.set_row_height(7, 1, 22.345)
    assert self.data_array.row_heights[7, 1] == 22.345
```

Listing 1: Example method from the `pyspread` project.

```
def test_set_row_height(self):
    """Unit test for set_row_height"""
    [INSERT]
```

Listing 2: Stripped example method from the `pyspread` project.

As a result of this *stripping process* we have the original test T_i and a generated test G_i. This allows us to compare to what a human programmer would have written in that context. Furthermore, because we leave the method signature (and method comment, if available) Copilot has some information of the CUT being targeted. We are effectively simulating writing tests using GitHub Copilot in an IDE.

2.5 With- and Without-Context

For every test invoked within an existing test suite (With-Context), we also invoke the test without an existing test (Without-Context). Meaning that all test files (except the test file of the original test) are deleted. Within the test file of the original test, all other test methods are deleted. This results in a single test file, with a single test method. Code imports, and helper/utility functions are kept.

2.6 Varying Test Method Comments

During the stripping process, the test method comment can be changed. We evaluate the usability of generations with varying test method comment strategies, to determine how a test method comment should be formulated to attain the most usable test generations. We use a smaller subset of O called M to evaluate method comment strategies. This subset M contains failing test pairs (T_i, G_i) belonging to the following projects: `django-multiurl`, `pyspread`, `click`, `exif`, and `python-crontab`. A failing test pair is a test pair for which the generation G_i is not a passing test or empty. We selected failing test pairs from only these five aforementioned projects to simplify the method comment formulation process.

We devise four method comment strategies which we evaluate. This means that for every pair in $(T_i, G_i) \in M$ we invoke four generations with a modified method comment, and investigate their usability. We define and demonstrate an example for each of the four strategies using the example test method in Listing 3.

```
def test_no_match(self):
    with self.assertRaises(urlresolvers.Resolver404):
        self.patterns_catchall.resolve('/eggs/and/bacon/')
```

Listing 3: Example method from the `django-multirurl` project.

(1) Minimal Method Comment. This type of comments provides a minimal description of a particular test method.

```
def test_no_match(self):
    """Test the resolve function"""
    [INSERT]
```

Listing 4: Test method with a Minimal Method Comment.

(2) Behavior-Driven Development Comment. This type of comment provides a Behavior-Driven Development scenario description of a particular test method. The basic structure of the formulation is as follows: "Given x when y then z."

```
def test_no_match(self):
    """Given that I resolve a URL
    when that URL does not match
    then an exception should be raised"""
    [INSERT]
```

Listing 5: Test method with a Behavior-Driven Development Comment.

(3) Usage Example Comment. This type of comments provides a code snippet containing a possible call of the CUT, as a usage example. The example does not need to relate to a specific scenario a test is testing, but only to the CUT. This approach is based on earlier LLM test generation work by Schäfer et al. [23].

```
def test_no_match(self):
    """example usage:

    url = urlresolvers.URLResolver(RegexPattern(r'^/'), [
        multiurl(
            url(r'^(\w+)/$', x, name='x')
        )
    ])
    url.resolve('/jane/')

    gives:

    ResolverMatch() object"""
    [INSERT]
```

Listing 6: Test method with a Usage Example Comment.

(4) Combined. Finally, we combine all aforementioned comment formulations in a single method comment.

[7] GitHub Copilot does not provide an official API, hence we manually use Copilot in an IDE.

($n = 53$)	With-Context	Without-Context
Passing Tests	24 (45.28%)	4 (7.55%)
Failing Generations	29 (54.72%)	49 (92.45%)
– Failing Tests	9 (16.98%)	10 (18.87%)
– Broken Tests	12 (22.64%)	38 (71.70%)
– Syntax Error	3 (5.66%)	11 (20.75%)
– Runtime Error	9 (16.98%)	27 (50.94%)
– Empty Generation	8 (15.09%)	1 (1.89%)

Table 2: Breakdown of all Copilot generations for $RQ_{1,2}$.

The method comments were manually formulated by the first two authors. They independently formulated the comments using two proper subsets of M, where an intersection of those two subsets was first discussed to come to a negotiated agreement on how to formulate each type of method comment.

2.7 Study Execution

Within this research we consider the first suggestion (generation) provided when GitHub Copilot is invoked, and do not consider the alternative suggestions (when they are available) to allow for a consistent and fair comparison.[8] Suggestions were manually invoked in the PyCharm IDE version 2022.2.4 with the GitHub Copilot plugin version 1.2.3.2385. All generations requested from Copilot in this study occurred between December 2022 and May 2023. All coverage computation was done using Coverage.py.[9]

3 RESULTS

Within this section we report the results for all the research questions. We combine the results of RQ_1 and RQ_2 to allow us to easily compare them. Idem for RQ_3 and RQ_4.

3.1 $RQ_{1,2}$: How usable is a test generated by GitHub Copilot *within* and *without* the context of an existing test suite?

In total 53 test pairs (T_i, G_i) were considered, forming the set O. All considered test pairs $(T_i, G_i) \in O$ were labelled with one or multiple code aspects. All labels created (with their definition) can be found in Table 3. The code aspect labels were iteratively created, we stopped sampling sets of tests when there were no more new code aspect labels that could be created (theoretical saturation).

In Table 2 we find that 54.72% (29 generated tests) of all generations With-Context and 92.45% (49 generated tests) of all generations Without-Context are failing generations. A failing generation is a generation G_i that is a failing, broken, or empty test generated by Copilot. We define a broken test as a test with either a syntax or runtime error. We discuss the usability aspect of these test generations in the following subsections.

3.1.1 Syntax and Runtime Correctness. In Table 2 we find that 22.64% of With-Context and 71.70% of Without-Context tests generations are broken tests. We have identified six reasons demonstrating why these syntax and runtime errors occur in broken tests, and we illustrate them with an example:

[8]Getting started with GitHub Copilot (Seeing alternative suggestions): https://docs.github.com/en/copilot/getting-started-with-github-copilot#seeing-alternative-suggestions, last visited October 25th, 2023.
[9]Coverage.py: https://github.com/nedbat/coveragepy, last visited October 25th, 2023.

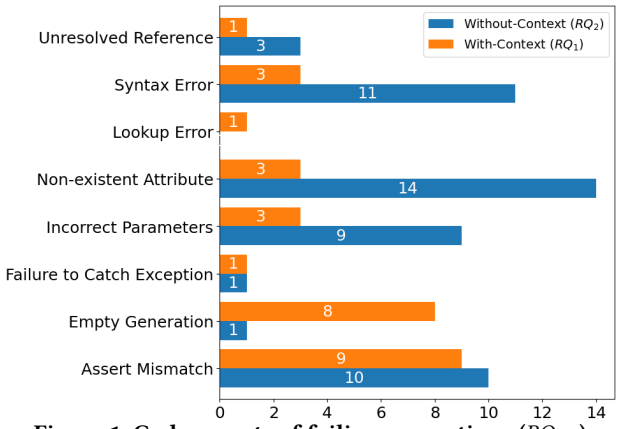

Figure 1: Code aspects of failing generations ($RQ_{1,2}$).

(1) Syntax Error. In all test generations containing a Syntax Error, both With- and Without-Context, the failing generation appears to be incomplete which results in a Syntax Error. Up to the last row of every generated test method with a Syntax Error all code is syntactically correct. In Listing 7 for example, the generated `test_captures_stdout_stderr` method fails because the last row of the generation is incomplete — it misses a closing quote and parenthesis.

```
1  def test_captures_stdout_stderr(self):
2      """Test capturing stdout and stderr from print
3      """
4      message = "This should be captured..."
5
6      buf = io.StringIO()
7      with stdout_err_redirector(buf):
8          print(message)
9          print(message, file=sys.stderr)
10
11     self.assertEqual(buf.getvalue(), message + '\n' + message
         + '\
```

Listing 7: Failing generation due to an Syntax Error. On row 11 the single quote and parenthesis should be closed.

Code Aspect Label	Definition
Assert Mismatch	Contains an assertion that evaluates to false.
Empty Generation	Received an empty generation from GitHub Copilot.
Incorrect Parameters	Uses keyword arguments (parameters) of a class or method incorrectly. Either by passing down inapplicable objects or values, or by passing down an incorrect number of arguments.
Syntax Error	The generated test contains a syntax error.
Non-existent Attribute	Uses an attribute of an object, but the attribute does not exist or is not subscriptable.
Unresolved Reference	Contains a reference to an object which does not exist in the namespace.
Failure to Catch Exception	Raises an exception which is not captured, but should be captured (as can be determined from the original test).
Lookup Error	Uses a key of an object, but the key does not exist.

Table 3: List of code aspect labels with their definition.

(2) Incorrect Parameters. Incorrect Parameters result in a failing generation when the keyword arguments (parameters) being passed down are not the expected parameters for a particular class or method. For example in Listing 8, the method `test_option_optional` is using a keyword argument `optional` in the creation of a class instance of an `Option` object. However, the keyword `optional` does not exist for an `Option` object. A `TypeError` is thrown noting the unexpected keyword argument (i.e., incorrect parameter). Another form by which Incorrect Parameters result in failing generations is when the parameters being used do exist, but are provided an incorrect value. For example, when an integer is expected but a string is given.

```
1 def test_option_optional():
2     cli = Command("cli", params=[Option(["-c"], optional=True
      )])
3     assert _get_words(cli, ["-c"], "") == []
4     assert _get_words(cli, ["-c"], "-") == ["--help"]
```

Listing 8: Failing generations due to an Incorrect Parameter. On row 2 the class `Option` does not have an argument `optional`.

(3) Non-existent Attribute. Non-existent Attribute occurs when an attribute of an object is being used, and the attribute does not exist or is not subscriptable (i.e., Copilot assumes an object contains other objects which can be accessed). For example in Listing 9, `test_modify_ascii_same_len` fails because the object `self.image` does not contain the attribute `_update_exif()`.

```
1 def test_modify_ascii_same_len(self):
2         """Verify that writing a same length string to an
      ASCII tag updates the tag."""
3         self.image.model = "Canon EOS 5D Mark III"
4         self.image._update_exif()
5         check_value(self, self.image.model, "Canon EOS 5D
      Mark III")
6         check_value(self, self.image._get_exif(),
      MODIFY_ASCII_SAME_LEN_HEX_BASELINE)
```

Listing 9: Failing generations due to an Non-Existent Attribute. The `_update_exif()` attribute does not exist for `self.image`.

(4) Unresolved Reference. In a generation with an Unresolved Reference, a reference is made to an object which does not exist. For example, the generated test method (see Listing 10) contains the usage of a class named `CronRange`. This results in a `NameError` stating that `CronRange` is not defined. Within the project there does exist a class named `CronRange`. However, it is not imported and thus does not exist in the namespace.

```
1 def test_18_range_cmp(self):
2     """Compare ranges"""
3     self.assertEqual(CronRange('*/6'), CronRange('*/6'))
4     self.assertNotEqual(CronRange('*/6'), CronRange('*/7'))
5     self.assertNotEqual(CronRange('*/6'), CronRange('*/6-7'))
```

Listing 10: Failing generations due to an Unresolved Reference. The `CronRange` object does not exist.

(5) Failure to Catch Exception. In a generation with a Failure to Catch Exception, an exception is raised which is meant to be caught, but the generation fails to do so, e.g., in Listing 11 the generated test method `test_cli` fails because a `SystemExit` exception is thrown by `parser.parse_args()` in some cases (this is a parametrized test).

```
1  def test_cli(argv, res):
2      """Test cli"""
3
4      with patch.object(sys, 'argv', argv):
5          parser = PyspreadArgumentParser()
6          args = parser.parse_args()
7
8          if res is not None:
9              assert args == res
10         else:
11             assert args is None
```

Listing 11: Failing generations due to an Failure to Catch Exception.

(6) Lookup Error. Similar to Non-existent Attribute, but instead a key or index value is being used which does not exist. For example, the generated test method `test_06_env_access` (see Listing 12) fails because the key value `CRON_VAR` does not exist in `self.crontab.env`. Another Lookup Error is when an out of range index value is used on a list or array.

```
1 def test_06_env_access(self):
2     """Test that we can access env variables"""
3     self.assertEqual(self.crontab.env['PERSONAL_VAR'], 'bar')
4     self.assertEqual(self.crontab.env['CRON_VAR'], 'fork')
5     self.assertEqual(self.crontab[0].env['CRON_VAR'], 'fork')
6     self.assertEqual(self.crontab[1].env['CRON_VAR'], 'spoon'
      )
7     self.assertEqual(self.crontab[2].env['CRON_VAR'], 'knife'
      )
8     self.assertEqual(self.crontab[3].env['CRON_VAR'], 'knife'
      )
9     self.assertEqual(self.crontab[3].env['SECONDARY'], 'fork'
      )
```

Listing 12: Failing generations due to an Lookup Error. The key value `self.crontab.env['CRON_VAR']` does not exist.

In Figure 1 we observe that tests generated Without-Context have an increase in overall occurrence of Non-existent Attributes (366.67%), Incorrect Parameters (200%), Syntax Errors (266.67%), and Unresolved References (200%) when compared to tests generated With-Context. Tests generated Without-Context have a decrease of 87.5% in Empty Generation occurrences. Failing generations, both With- and Without-Context, fail for reasons such as having Non-existent Attributes and Incorrect Parameters. These reasons for failing generations are particularly prominent in generations Without-Context. This suggests that Copilot does not consider the CUT, but only other test methods in its context. Due to the closed-source nature of Copilot, the exact context used to prompt generations can only be speculated. It may be due to the limited context length (4096 tokens) of Codex [8, 25].

Observation I

Our findings suggest that GitHub Copilot does not consider the CUT when invoking Copilot to generate tests written in files separate from the code.

3.1.2 Passing. In Table 2 we find that 45.28% and 7.55% of the generated tests were passing for With- and Without-Context (respectively). Our manual inspection of passing tests With-Context reveals that some passing tests appear to "mimic" tests in their direct test context (in the same test file). To demonstrate the mimicking behavior we show a generated test in Listing 13 and a test in its direct context in Listing 14.

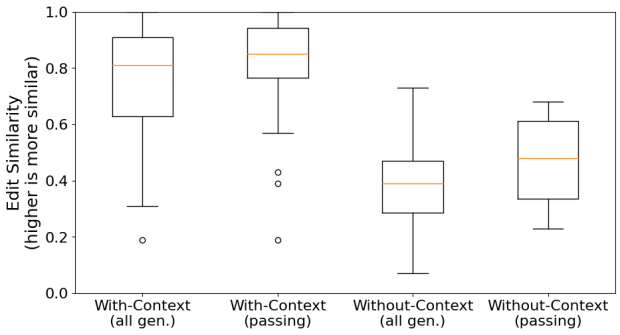

Figure 2: Box plot of the edit similarities between every G_i and the most similar test T in the direct context of G_i for $RQ_{1,2}$. An edit similarity of $s(x, y) = 1$ would indicate an exact copy. Empty generations are excluded.

```
def test_option_optional():
    cli = Command("cli", params=[Option(["-c"], optional=True
     )])
    assert _get_words(cli, ["-c"], "") == []
    assert _get_words(cli, ["-c"], "-") == ["--help"]
```

Listing 13: A test generated by Copilot.

```
def test_option_count():
    cli = Command("cli", params=[Option(["-c"], count=True)])
    assert _get_words(cli, ["-c"], "") == []
    assert _get_words(cli, ["-c"], "-") == ["--help"]
```

Listing 14: A highly similar test in the test context of Listing 13.

The generated test in Listing 13 appears to be mimicking the test in Listing 14. To further capture this mimicking behavior we compute the edit similarity between the generated test and every other test in its direct test context. We compute the edit similarity for a given test T in the direct test context G_i as follows:[10] $s(T, G_i) = 1 - \frac{d(T,G_i)}{\max(|T|,|G_i|)}$ where $d(x, y)$ is the Levenshtein distance between x and y. For each generated test G_i we find the *most* similar test T in the direct context of G_i, resulting in a set of edit similarities consisting of the edit similarity value between every G_i (excluding empty generations) and the most similar test T in the direct context of G_i. In Figure 2 we have a box plot summarizing all edit similarities for both With- and Without-Context. Overall, we can see that generations G_i With-Context are similar to some test T in the direct context of G_i. This effect is particularly pronounced for passing tests, which are even more similar. It appears that Copilot is mimicking tests in the direct context for its test generations. Without-Context, Copilot cannot mimic tests in the direct context, and hence the generations are less similar.

Observation II

Our findings suggest that GitHub Copilot depends on the direct test context (i.e., the test file in which Copilot is invoked) to produce passing tests. Often, mimicking a neighboring test method when generating a test method.

Moving on, 17.92%[11] of tests generated (combining both With- and Without-Context) are failing tests. To determine correct asserts is a hard problem in itself: the *test oracle problem* refers to the problem of determining correct and incorrect behavior, so that correct asserts can be formulated [6]. We consider a generation that is a passing test better than a generation that is a failing test, as it arguably requires fewer modifications to be used in a test suite. However, failing tests can actually reveal new faults. Perhaps the assertions generated in the test are correct, and the CUT is faulty. Likewise, a passing test may be conforming faulty behavior of CUT. Nonetheless, we focus on the usability of generations, and not their fault-finding ability. We assume that the CUT is correct as the original test is passing. Although it is important to note that whether a generation is passing or failing does not necessarily say anything about correctness or incorrectness of the CUT.

Furthermore, we find that 32.07% of tests generated With-Context are failing or empty tests. We find a similar percentage of failing tests for Without-Context generations, but a 87.5% decrease in occurrences of empty tests. Failing tests are those labeled with Assert Mismatch, Empty Generation occurs when Copilot returns nothing:

(1) Assert Mismatch. Copilot is not able to determine the expected value of one or multiple assertions for the CUT. For example the generated `test_handle_bad_attribute` method is asserting whether an `AttributeError` is raised where the error messages should match the following string: `"unknown image attribute fake_attribute"`. Copilot however fails to generate the correct string (see Listing 15), which results in an Assert Mismatch (failing test). In this case, Copilot did not have sufficient information to correctly determine the string to be matched on.

```
def test_handle_bad_attribute():
    """Verify that accessing a nonexistent attribute raises
     an AttributeError."""
    with open(
        os.path.join(os.path.dirname(__file__), "grand_canyon
     .jpg"), "rb"
    ) as image_file:
        image = Image(image_file)

    with pytest.raises(AttributeError, match="image does not
     have attribute"):
        image.fake_attribute
```

Listing 15: Failing generation due to an Assert Mismatch. The string being matched on (row 8) should be: `"unknown image attribute fake_attribute"`.

(2) Empty Generation. The underlying reason for Copilot not returning a generation can only be speculated; it could be caused by context length limit of Codex [25]. The prompt being formulated by Copilot based on the code context may be too long for the model.

3.1.3 Coverage. We compute the branches covered[12] by the original and generated test of every passing test pair $(T_i, G_i) \in O$ for both With- (24 tests) and Without-Context (4 tests). Table 4 provides an overview of all passing tests generated With- and Without-Context and their coverage data. Recall that for the coverage usability aspect we consider generations G_i covering fewer of the same branches than the original test T_i as negatively impacting the usability, and thus being less *suitable* with respect to the human-written test T_i; this suitability is captured by the Branch Overlap Ratio (BOR).

[10] This approach is based on the work of Schäfer et al. [23] and Ippolito et al. [23] who employed edit similarity (or alike) to determine how similar a test was to a generated test in the context of LLM memorization.

[11] $(9 + 10)/(53 + 53) = 17.92\%$

[12] The lines covered was also considered initially but yielded similar results.

#	Test Name	Diff. Covered Branches (T_i, Gi)	Branch Overlap Ratio
1	test_option_custom_class_reusable	+7.38%	1.0
2	test_show_true_default_boolean_flag_value	0.0	1.0
3	test_resolve_match_first	0.0	1.0
4	test_resolve_match_last	0.0	1.0
5	test_resolve_match_middle	0.0	1.0
6	test_resolve_match_path_brand	0.0	1.0
7	test_list_all	0.0	1.0
8	test_main_no_processes_long	0.0	1.0
9	test_data_access	0.0	1.0
10	test_on_markup_renderer_pressed	0.0	1.0
11	test_rgb2qimage	0.0	1.0
12	test_insertTable	0.0	1.0
13	test_row	0.0	1.0
14	test_rgb666	0.0	1.0
15	test_set_row_height	0.0	1.0
16	test_get_absolute_access_string	0.0	1.0
17	test_04_number	0.0	1.0
18	test_find_list	0.0	1.0
19	test_06_clear	0.0	0.98
20	test_21_slice_special	0.0	0.8
21	test_progressbar_item_show_func	-1.5%	0.98
22	test_remove_layer	-11.36%	0.89
23	test_09_removal_during_iter	-16.9%	0.81
24	test_command	-19.85%	0.85
1	test_get_absolute_access_string	0.0%	1.0
2	test_09_removal_during_iter	-4.23%	0.78
3	test_get_context_objects_missing	-15.89%	0.87
4	test_handle_bad_attribute	-95.92%	0.06

Table 4: Overview of coverage data for all passing test pairs With- and Without-Context ($RQ_{1,2}$). Tests below the bold line are tests generated Without-Context.

When considering passing tests With-Context, we find that 17 of the 24 (rows 2–18) generated passing tests cover the same branches as their human-written counterpart T_i. Only one generated test (row 1) covers the same and more new branches. The remaining six generated tests (rows 19–24) cover strictly fewer branches and/or cover new branches. Hence, most passing tests generated by Copilot With-Context do not cover fewer branches than the original test T_i, which positively impacts the usability. Without-Context, we find that only one generated test (row 1) covers the same branches as their human-written counterpart. The remaining generated tests (rows 2–4) cover fewer and/or new branches. This indicates that tests generated Without-Context, even if passing, are less suitable.

Observation III

Our findings suggest that 70.1% of all passing tests generated by GitHub Copilot within the context of an existing test suite cover the exact same branches as the same tests written by humans.

Identical branches being exercised in the test cases (rows 2–18) can be partly explained due to most passing tests mimicking an existing test within the direct test context. As a result of this mimicking, Copilot may produce a generation that is highly similar to the original test. This is for example the case when the original test T_i has tests in its direct test context which are similar, but each one exercises a slightly different test scenario (such as the case for repetitive tests with similar method signatures). Copilot mimics one of the neighboring tests to produce its generation G_i, which ends up being (nearly) identical to the original test T_i. This explains why the majority of passing tests generated from Copilot in Table 4 cover the exact same branches as the original test T_i.

3.1.4 Research Answers ($RQ_{1,2}$). Less than half (45.28%) of all generations invoked within the context of an existing test suite are passing tests; 54.72% of generations are failing, broken, or empty tests. Most passing tests cover the same branches as their human-written counterpart. However, due to the majority of generations being failing generations, the overall usability of Copilot's test generation ability is negatively affected, as close to half of all generations will require modification before being used in a test suite.

Answer to RQ_1

How usable is a test generated by GitHub Copilot *within* the context of an existing test suite? The usability of tests generated by GitHub Copilot within the context of an existing test suite is poor. Most tests generated will need to be modified.

A minority (7.55%) of all generations invoked without the context of an existing are passing tests. The majority (92.45%) of generations are failing, broken, or empty tests. Passing tests mostly cover fewer of the same branches as their human-written counterparts. Due to the majority of generations being failing generations, and passing tests covering fewer branches, the overall usability of Copilot's test generation ability is negatively affected, as most generations will require modification before being usable in a test suite.

Answer to RQ_2

How usable is a test generated by Copilot *without* the context of an existing test suite? The usability of tests generated by GitHub Copilot without the context of an existing test suite is very poor. Almost all tests generated will need to be modified.

3.2 $RQ_{3,4}$: How should a test method comment be formulated to attain a usable test generation from GitHub Copilot *within* and *without* the context of an existing test suite?

In total, we select 23 tests for which the respective generation was found to be failing in $RQ_{1,2}$. These test pairs form the set M. Recall that for $RQ_{3,4}$ we formulate four test method comment strategies: **Minimal Method Comment (MMC)**, **Behavior-Driven Development Comment (BDDC)**, **Usage Example Comment (UEC)**, and **Combined Comment (CC)**. We are interested which of these four method comment strategies result in the most usable test generations. We compare each method comment strategy using the aspects of usability and discuss the best performing ones.

For each test in T_i in M, we apply the four method comment strategies and then invoke Copilot with the adjusted method comment. This results in four generations for each test T_i, which we denote as $(T_i, G_{i,k})$ where k indicates which of the four method comment strategies was used (e.g., $G_{i,MMC}$). All pairs $(T_i, G_{i,k})$ are assigned code aspect labels using the labels defined in Table 3.

Table 5 presents a breakdown of all test generations for each method comment strategy, both With- and Without-Context. Figure 3 shows a summary of code aspects of failing generations for

With-Context	(n = 23) MMC	(n = 23) BDDC	(n = 23) UEC	(n = 23) CC
Passing Tests	5 (21.74%)	6 (26.09%)	**8 (34.78%)**	6 (26.09%)
Failing Generations	18 (78.26%)	17 (73.91%)	15 (65.22%)	17 (73.91%)
– Failing Tests	**8 (34.78%)**	7 (30.43%)	**8 (34.78%)**	**8 (34.78%)**
– Broken Tests	7 (30.43%)	7 (30.43%)	4 (17.39%)	6 (26.09%)
– Syntax Error	1 (4.35%)	1 (4.35%)	1 (4.35%)	1 (4.35%)
– Runtime Error	6 (26.09%)	6 (26.09%)	3 (13.04%)	5 (21.74%)
– Empty Generations	3 (13.04%)	3 (13.04%)	3 (13.04%)	3 (13.04%)
Without-Context	**MMC**	**BDDC**	**UEC**	**CC**
Passing Tests	4 (17.39%)	3 (13.04%)	**5 (21.74%)**	**5 (21.74%)**
Failing Generations	19 (82.61%)	20 (86.96%)	18 (78.26%)	18 (78.26%)
– Failing Tests	7 (30.43%)	6 (26.09%)	7 (30.43%)	**11 (47.83%)**
– Broken Tests	12 (52.17%)	14 (60.87%)	11 (47.83%)	7 (30.43%)
– Syntax Error	1 (4.35%)	4 (17.39%)	1 (4.35%)	1 (4.35%)
– Runtime Error	11 (47.83%)	10 (43.48%)	10 (43.48%)	6 (26.09%)
– Empty Generations	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)

Table 5: Breakdown of all Copilot generations for $RQ_{3,4}$ With- and Without-Context. Cells in bold highlight the largest number of passing or failing tests in the row.

both With- and Without-Context for all considered method comment strategies. We find that the application of any one of these method comment strategies resulted in more passing tests than not following any of these method comment strategies. We compare the method comment strategies in the following subsections.

3.2.1 Syntax and Runtime Correctness. Table 5 shows that the **Usage Example Comment** strategy produces generations with the lowest number of broken tests With-Context. We further find that With-Context the number of failing tests for the **Usage Example Comment** and **Combined Comment** strategy is the same, but Without-Context the **Combined Comment** has more failing tests and fewer broken tests than the **Usage Example Comment** despite having the same number of passing tests. This indicates that a **Combined Comment** strategy would be more usable than a **Usage Example Comment** strategy when generating tests Without-Context, as the **Combined Comment** strategy produces fewer broken tests while having the same number of passing tests as the **Usage Example Comment** strategy Without-Context. We consider broken tests to be less usable than failing tests. Nonetheless, it will require more effort to formulate a **Combined Comment** than a **Usage Example Comment**. Thus, in practice, a **Usage Example Comment** would yield similar results but with less effort.

Minimal Method Comment and **Behavior-Driven Development Comment** produce more broken tests and fewer passing tests than either the **Combined Comment** or **Usage Example Comment** strategy. This suggests that including a usage code example as part of the method comment yields more passing tests.

Observation IV

Our findings suggest that test method comment strategies that include a code usage example result in more passing test generations than test method comment strategies without a code usage example.

Figure 3 presents which type of syntax or runtime errors occur for all the different method comment strategies. We find the same reasons as found in $RQ_{1,2}$ apply here, but with different distributions per comment strategy. We also note that independent of the comment strategy applied a Without-Context generation results in an increase in broken tests. The overall occurrence of Incorrect Parameters and Unresolved References increases for generations Without-Context independent of comment strategy.

3.2.2 Passing. In Table 5 we see that for With-Context the **Usage Example Comment** yields the most passing tests. For Without-Context, both **Usage Example Comment** and **Combined Comment** produce the same number of passing tests. Additionally, we find that Copilot is mimicking existing tests in the direct test context (see Section 3.1.2), although the effect is less pronounced overall for passing tests across all method comment strategies. Furthermore, we observe that generations Without-Context produce no empty tests independent of method comment strategy, this is similar to our earlier finding in $RQ_{1,2}$: Generations made Without-Context are less likely to be empty tests. This could be caused by context length limit of Codex [25]. The prompt being formulated by Copilot based on the code context may be too long for the model.

Observation V

Our findings suggest that GitHub Copilot is less likely to produce an empty test generation when there is no test code context.

3.2.3 Coverage. We compute the branches covered for the original T_i and generated $G_{i,k}$ for every pair $(T_i, G_{i,k})$ where $G_{i,k}$ is a passing test. Recall that for the coverage usability aspect we consider covering fewer of the same branches as the original test T_i as negatively impacting usability. The ratio of the branches covered by the original test T_i and the generated test $G_{i,k}$ is captured using the Branch Overlap Ratio. Hence, we compare the different method comment strategies by computing the average BOR. Generations which are failing have a BOR of 0 by definition. over all test generated for each method comment strategy. From Table 6 we observe that the **Usage Example Comment** yields the highest average BOR when considering all tests generated With-Context compared to other strategies. Likewise, we find that the **Combined Comment** strategy produces test generations with the highest average BOR for Without-Context generations.

3.2.4 Research Answers ($RQ_{3,4}$). We find that the **Usage Example Comment** strategy produces the most passing tests (34.78%) and the least number of broken tests (17.39%) in the context of an existing test suite. Furthermore, the **Usage Example Comment** produces test generations with the highest average ratio of covered branches overlapping with their human-written counterparts.

	With-Context				Without-Context			
	(n = 5) MMC	(n = 6) BDDC	(n = 8) UEC	(n = 6) CC	(n = 4) MMC	(n = 3) BDDC	(n = 5) UEC	(n = 5) CC
Avg. BOR	0.21	0.25	**0.34**	0.26	0.17	0.13	0.2	**0.21**

Table 6: Overview of the average Branch Overlap Ratio for each method comment strategy. Cells highlighted in bold indicate the highest average Branch Overlap Ratio in the row.

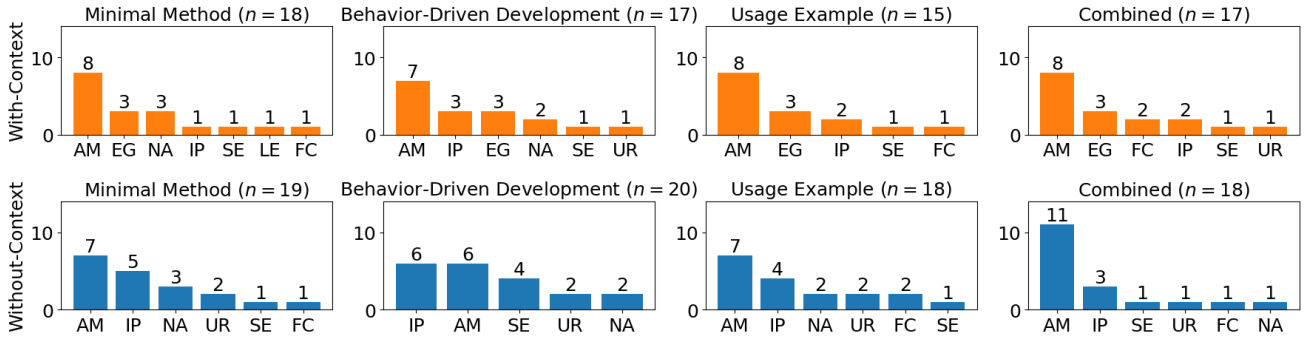

Figure 3: Summary of code aspects of failing generations for each method comment strategy for both With- and Without-Context ($RQ_{3,4}$). AM = Assert Mismatch, EG = Empty Generation, NA = Non-existent Attribute, IP = Incorrect Parameters, SE = Syntax Error, LE = Lookup Error, FC = Failure to Catch Exception, and UR = Unresolved Reference.

Answer to RQ_3

How should a test method comment be formulated to attain a usable test generation from GitHub Copilot *within* the context of an existing test suite? A test method comment should be formulated with a usage example to attain a usable generation within the context of an existing test suite.

Likewise, we find that the **Combined Comment** strategy produces the most passing tests (21.74%) and the least number of broken tests (30.43%) without the context of an existing test suite. Furthermore, the **Combined Comment** produces test generations with the highest average ratio of covered branches overlapping with their human-written counterparts.

Answer to RQ_4

How should a test method comment be formulated to attain a usable test generation from GitHub Copilot *without* the context of an existing test suite? A test method comment should be formulated as a comment combining instructive natural language with a code usage example to attain a usable generation without the context of an existing test suite.

4 DISCUSSION

Within this section we discuss the implications of our findings for three target groups: practitioners, researchers, and for the GitHub Copilot system itself.

Implications for Practitioners. Practitioners invoking GitHub Copilot (without modifying the method comment) in their software testing efforts, will find themselves applying manual modifications to GitHub Copilot's generations, even when applying GitHub Copilot within an existing test file with tests (see Answer to RQ_1). In that case, GitHub Copilot tends to mimic other tests for its generations (see Observation II). Our findings suggest that **invoking GitHub Copilot with the intention of mimicking an existing test is more likely to result in usable generation**. Without any existing test context, GitHub Copilot hardly provides any usable generation (see Answer to RQ_2). Nonetheless, practitioners can affect the generations by modifying the context. When you have an existing test context, we observe that **providing a code usage example within the test method before invoking GitHub Copilot will yield more usable generations** (see Answer to RQ_3). When there is no existing test context, a method comment providing both a natural language description and a usage example (**Combined Comment**) will yield more usable generations (see Answer to RQ_4). However, in practice, a **Usage Example Comment** will provide similar results to a **Combined Comment** with less effort (because it takes more time to formulate a **Combined Comment**).

Implications for Researchers. Previous research on test generation using LLMs demonstrated the value of including some form of a test code example when prompting the model [5, 23]. Generations from GitHub Copilot mimic their surrounding test context for its generations (see Observation II), essentially treating the surrounding test context as test code examples. Without that context, GitHub Copilot is unable to mimic and thus provides fewer usable generations (see Answer to RQ_2). Furthermore, we find that method comments containing a code usage example result in more passing tests, independent of the test context (see Observation IV). Hence, we hypothesize that **test code examples are useful for generating tests using LLMs**.

Nonetheless, GitHub Copilot's generations are often broken tests, even when test code context is present (see Answer to RQ_1). Broken tests frequently contain runtime errors such as Unresolved References or Non-existent Attributes. In these cases, **GitHub Copilot is "hallucinating" references, object attributes, or alike; these references or attributes do not actually exist**. Hallucination is a larger challenge within language generation models and refers to generations that are "nonsensical or unfaithful to the provided source content" [19]. Further research should investigate how these hallucinations can be mitigated when generating tests.

Implications for GitHub Copilot. Our findings suggest that **GitHub Copilot does not consider the code under test (CUT) when generating tests** (see Observation I). We hypothesize that Copilot does not send the CUT as part of the prompt it uses for generating its suggestion. Furthermore, while we do not know the exact model which powers Copilot, we do know that the Codex model has limited context length (4096 tokens), and that Copilot uses a model which descends from Codex [8]. Thus, we assume that it is not possible to include all CUT as part of the prompt for any reasonably sized project. One could **modify the prompts of the GitHub Copilot system such that it includes the relevant CUT, as much as possible, whenever GitHub Copilot is invoked for generating a test method**. Such a system could, e.g., use static

analysis to narrow down the relevant CUT based on code imports/references made in the direct test context, or possibly even the file or method name, and then include the CUT as part of the prompt.

5 THREATS TO VALIDITY

Within this section we discuss the primary threats to validity.

Internal Threats. The formulations of the different method comments are not unique for a given method. For example, every person will likely write a (slightly) different Behavior-Driven Development Comment for the same method. We partially mitigate this by ensuring method comment formulations were independently formulated by two contributors of this research. Finally, code aspect labels were iteratively created by manual inspection of Copilot's test generations. Despite careful labelling the labels are subject to human error. This however likely does not impact the results and findings overall as we investigated and labelled a large number of generations.

External Threats. While we have considered in total 290 test generations from GitHub Copilot, they all stem from 53 test methods sourced from seven open-source projects. All the selected open-source projects use the English language for their (code) documentation, and are focused on a diverse, but limited set of domains. The findings may not be generalizable to other domains. In addition, including more projects (which would lead to more generations) could strengthen the confidence of the findings presented. Furthermore, we focus on the Python programming language, and do not include other programming languages as part of our analysis.

Threats to Reproducibility. Due to GitHub Copilot's proprietary closed-source nature, the LLM underlying Copilot may change in the future. This can result in different code generations than those documented and investigated in this research, in turn impacting the reproducibility of the results. To mitigate this, we include all generations investigated in this study in our replication package [17].

6 RELATED WORK

Siddiq et al. [25] investigate the Java test generation ability of Codex [8], CodeGen [21], and OpenAI's GPT-3.5 [9] LLM. They construct varying prompt scenarios, e.g., containing the full code of the class under test, with code comments, etc. They find that across all models, numerous generated test were not compilable, even after they employed rule-based repairs to fix these generated test [25]. Overall, they find that the LLMs perform worse than EvoSuite in terms of line and branch coverage, and number of passing tests. Similarly, Bareiß et al. [5] investigate the Java test generating ability of Codex [8], among other tasks. They generate tests for 18 Java methods using a prompt for each method consisting of helper functions, an example method with a respective test, and the method under test. In particular, they find that Codex achieves higher coverage than Randoop for the 18 Java methods. Furthermore, they report that "suitable" examples are key for the Codex model to make "effective predictions" [5]. Similar to Siddiq et al. [25] and Bareiß et al. [5] we investigate the test generation ability of a LLM; while they directly employ the Codex LLM for their test generations, we use GitHub Copilot. Furthermore, Siddiq et al. [25] investigate test smells in generations to assess their quality, whereas we consider usability aspects.

Yuan et al. [29] investigates the test generation ability of ChatGPT, and develops ChatTester, an approach that uses prompt refining (meaning that the prompt is automatically refined based on whether the previous prompt generated a passing test) to repair broken tests. The initial prompt includes the focal method (CUT), relevant code imports, and a natural language description instructing ChatGPT to generate a test for the focal method. They find that nearly 42.1% of all tests generated by ChatGPT fail due to compilation or execution errors. Our work differs in at least two ways: (1) Yuan et al. [29] employ ChatGPT instead of GitHub Copilot (which directly integrates in IDE) for test generations (2) they use a form of prompt refining, whereas we only consider the generation outputted by GitHub Copilot.

7 CONCLUSION

We investigated the usability of in total 290 tests generated by GitHub Copilot in several scenarios: with- and without an existing test suite, and with four different method comment strategies. We have defined several usability aspects to investigate the generations.

Firstly, we find that 45.28% of test generated by Copilot within an existing test code context are passing tests, containing no syntax or runtime errors. The majority (54.72%) of generated tests within an existing test code context are failing, broken, or empty tests. We observe that tests generated within an existing test code context often mimic existing test methods. In part due to this mimicking effect, passing tests generated by Copilot within existing test code context often cover the exact same branches as their human-written counterpart, which indicates that these generations are suitable.

Secondly, we find that tests generated by Copilot without an existing test code context are less usable, with 92.45% test generations failing, being broken, or empty tests. Only 7.55% of tests generated without existing test code context were passing, and most of them covered fewer branches than their human-written counterparts.

Additionally, we study how test method comments influence the usability of test generations. We find that test method comments using a code usage example produced the most usable test generations when invoking Copilot within an existing test code context. Without existing test code context, a comment combining instructive natural language with a code usage example yielded the most usable test generations.

Future work. We intend to investigate Copilot's output in test suites written in different programming languages. In particular, statically typed programming languages might produce different reasons for failing generations. Furthermore, a newly released blog article[13] from GitHub states that newer versions of GitHub Copilot consider other code files which are open in the IDE for its generations. Future research should consider how opening certain files in the IDE impacts the usability of Copilot's test generations.

ACKNOWLEDGMENTS

This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032) and conducted as part of Khalid El Haji's master thesis [16].

[13]"How GitHub Copilot is getting better at understanding your code": https://github.blog/2023-05-17-how-github-copilot-is-getting-better-at-understanding-your-code/, last visited October 25th, 2023.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4998–5007.

[2] Naser Al Madi. 2022. How Readable is Model-generated Code? Examining Readability and Visual Inspection of GitHub Copilot. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[3] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[4] Andrea Arcuri. 2018. An Experience Report on Applying Software Testing Academic Results in Industry: We Need Usable Automated Test Generation. *Empirical Softw. Engg.* 23, 4 (aug 2018), 1959–1981.

[5] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *ArXiv* abs/2206.01335 (2022).

[6] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[7] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 179–190.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Xuanting Chen, Junjie Ye, Can Zu, Nuo Xu, Rui Zheng, Minlong Peng, Jie Zhou, Tao Gui, Qi Zhang, and Xuanjing Huang. 2023. How Robust is GPT-3.5 to Predecessors? A Comprehensive Study on Language Understanding Tasks. *arXiv preprint arXiv:2303.00293* (2023).

[10] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling Readability to Improve Unit Tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 107–118.

[11] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 201–211.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[13] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*. ACM, 416—-419.

[14] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 24, 4, Article 23 (sep 2015), 49 pages.

[15] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. 2018. An Empirical Investigation on the Readability of Manual and Generated Test Cases. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 348–3483.

[16] Khalid El Haji. 2023. *Empirical Study on Test Generation Using GitHub Copilot*. Master's thesis. Delft University of Technology.

[17] Khalid El Haji. 2023. *Empirical Study on Test Generation Using GitHub Copilot — Replication Package*. https://doi.org/10.5281/zenodo.8025746

[18] Daphne Ippolito, Florian Tramèr, Milad Nasr, Chiyuan Zhang, Matthew Jagielski, Katherine Lee, Christopher A Choquette-Choo, and Nicholas Carlini. 2022. Preventing Verbatim Memorization in Language Models Gives a False Sense of Privacy. *arXiv preprint arXiv:2210.17546* (2022).

[19] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *Comput. Surveys* 55, 12 (2023), 1–38.

[20] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. ACM, 1—-5.

[21] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[22] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA-Companion)*. ACM, 815—-816.

[23] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive Test Generation Using a Large Language Model. *arXiv preprint arXiv:2302.06527* (2023).

[24] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. 2019. On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. IEEE, 121–125.

[25] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. *arXiv preprint arXiv:2305.00418* (2023).

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[27] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 8696–8708.

[28] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764* (2023).

[29] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv preprint arXiv:2305.04207* (2023).

[30] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity Assessment of Neural Code Completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS)*. ACM, 21—-29.