Empirical Study on Test Generation Using GitHub Copilot

Master's Thesis

Khalid El Haji

Empirical Study on Test Generation Using GitHub Copilot

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Khalid El Haji
born in Purmerend, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Empirical Study on Test Generation Using GitHub Copilot

Author: Khalid El Haji
Student id: 4733290

Abstract

Writing unit tests is a crucial task in the software development lifecycle, ensuring the correctness of the software developed. Due to its time-consuming and laborious nature, it is, however, often neglected by software engineers. Numerous automatic test generation tools have been devised to ease unit testing efforts, but these test generation tools produce tests that are typically difficult to understand. Recently, Large Language Models (LLMs) have shown promising results in generating unit tests and in supporting other software engineering tasks. LLMs are capable of producing natural-looking (human-like) source code and text. In this thesis, we investigate the usability of tests generated by GitHub Copilot, a proprietary closed-source code generation tool that uses a LLM for its generations and integrates into well-known IDEs. We evaluate GitHub Copilot's test generation abilities both within and without an existing test suite. Furthermore, we also evaluate the impact of different code commenting strategies on test generations, both within and without an existing test suite. We devise aspects of usability to investigate GitHub Copilot's test generations. In total, we investigate the usability of 290 tests generated by GitHub Copilot. Our findings reveal that *within* an existing test suite, approximately 45.28% of the tests generated by Copilot are passing tests. The majority (54.72%) of generated tests in an existing test suite are failing, broken, or empty tests. Furthermore, tests generated by Copilot *without* an existing test suite are less usable compared to those generated within an existing test suite. The vast majority (92.45%) of these test generations are failing, broken, or empty tests. Only 7.55% of tests generated without an existing test suite were passing, and most of them provided less branch coverage when compared to human-written tests. Finally, we find that tests using a code usage example comment resulted in the most usable generations within an existing test suite. In contrast, when there is no existing test suite, a comment combining instructive natural language combined with a code usage example yielded the most usable test generations.

Thesis Committee:

Chair:	Prof. Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	C.E. Brandt, M.Sc., Faculty EEMCS, TU Delft
Committee Member:	Dr. J. Yang, Faculty EEMCS, TU Delft

Preface

My time has come, and with this final thesis I conclude my studies in the field of computer science at Delft University of Technology. The last eight months I have had the privilege and honor to work under the Software Engineering Research Group researching test generation using modern generative language models. Writing tests is the bane of every software engineer, in part due to its laborious nature. If we can relieve software engineers from its laborious nature, we make software engineers more likely to test. Because as tedious as writing tests is, testing is still important.

I would like to thank Carolin Brandt for her excellent guidance and feedback during the entire thesis process. I am very grateful to have had you as my daily supervisor. Furthermore, I would like to thank the principal supervisor Andy Zaidman for his critical feedback and valuable insights. Finally, I would like to thank my friends and family for supporting me throughout this time.

Khalid El Haji
Delft, the Netherlands
June 11, 2023

Contents

List of Figures

Chapter 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. Numerous tools have been created to automatically generate unit tests with the goal of achieving a high structural coverage [27]. Tools such as EvoSuite and Randoop are two well-known and researched tools for automatically generating tests [14, 24]. These automatic unit test generation tools rely on techniques that employ a form of automated white-box test generation, where the primary objective is to maximize structural code coverage. Aspects such as readability and understandability of generated unit tests are not of primary concern [15]. This results in poor readability and understandability of the generated unit tests [16, 12]. 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, 15]. Automated test generation is thus still far from being widely adopted in the industry to assist in software engineering tasks [4].

GitHub Copilot[1] is a commercial code generation tool that integrates within existing popular IDEs (such as Visual Studio Code[2] or IntelliJ[3]). Copilot uses a Large Language Model (LLM) to produce code suggestions (henceforth, called generations) based on comments and code context. Large Language Models (LLMs) have shown promising results in numerous software engineering tasks, such as programming language translation [31], code completion [13], and code summarization [1]. LLMs are capable of "producing natural-looking completions for both natural language and source code" [26]. Fittingly, research on Copilot's generations has shown that Copilot produces readable and understandable generations, with similar complexity to human-written code [2, 22].

We hypothesize that GitHub Copilot can make writing tests a less time-consuming and tedious act for software engineers. However, this ultimately depends on how usable gener-

[1]GitHub Copilot: `https://github.com/features/copilot`
[2]Visual Studio Code `https://code.visualstudio.com/`
[3]IntelliJ: `https://www.jetbrains.com/idea/`

ated tests are. After all, 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. Within this thesis we attempt to find out how useable the generated tests from Copilot are, and how software engineers can use Copilot to generate more useable tests.

1.1 Research Goals

To evaluate the usability of GitHub 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. Some generations are more usable than other generations. For example, a generation with multiple runtime errors would be less usable than a generation with no runtime errors. In particular, we consider the following aspects for usability:[4]

Syntactic Correctness The generated test should not contain syntax errors.

Runtime Correctness The generated test should not give errors at runtime due to, for example, passing incorrect parameters to a method or alike.

Passing The generated test should be a non-empty passing test.

Non-Triviality The generated test should contain assertions and the assertions are not tautological assertions.

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 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 research questions:

[4]These aspects are partially based on earlier work by Schäfer et al. [26] and Xie et al. [32] who separately devised ways to evaluate the "quality" of tests generated by a LLM.

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?**

1.2 Research Approach

To answer our research questions, we conduct an empirical study on tests sampled from open-source Python projects. We consider projects from both GitHub and GitLab to mitigate potential training bias (the underlying model of GitHub Copilot is trained on open-source GitHub projects [9]). For each sampled test method, we strip the test method content/body leaving only the test method signature and comment. See Listing 1.1 for an example of a test stripped method.

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

Listing 1.1: Example stripped test method from the `pyspread` project. Note that the test method signature (row 1) and method comment (row 2) stay, but the test method body (row 3) has been stripped. They keyword `[INSERT]` indicates the place from which GitHub Copilot is invoked.

We then use Copilot to “regenerate” the test method body, which results in a test generation. We generate the test both within the existing test suite of the sampled test (With-Context), and without the test suite (Without-Context). We then document all aforementioned usability aspects for each generated test for both With- and Without-Context. Furthermore, we select a subset of the sampled tests to evaluate four different types of test method comment strategies. We manually formulate the four different method comments for each method in the subset, and then use Copilot to regenerate a test with each of the different method comments (for example, the method comment in Listing 1.1 is modified). Similarly, we document the usability aspects of generations produced using the different method comment strategies for both With- and Without-Context. Furthermore, we make a replication package available with all documented tests and scripts which we used to analyze the tests [17].

1.3 Structure of Thesis

The remainder of the thesis is structured as follows: Chapter 2 discusses background information and related work on test generation. In Chapter 3 we cover how we set up our study design and which choices we made. Then in Chapter 4 we present the results of the research question we are investigating, where we cover each usability aspect. In Chapter 5 we discuss the results and observations, answer the research questions, and also discuss the threats to validity. Finally, Chapter 6 concludes the thesis findings and provides directions for future research.

Chapter 2

Background and Related Work

Within this chapter we provide background information and an overview of research relating to our goal of evaluating GitHub Copilot's test generation abilities. We first provide background information, and then delve into more recent work employing and evaluating LLMs for test generation.

2.1 Traditional Test Generation Tools

There have been numerous tools developed with the aim of achieving high structural coverage by automatically generating unit tests [27]. Two widely recognized and extensively researched tools for this purpose are EvoSuite [14] and Randoop [24], which are considered to be state-of-the-art unit test generation tools [27]. These automatic unit test generation tools rely on search-based optimization and feedback-directed techniques (respectively) for test generation. The primary objective is to maximize structural code coverage of the target code under test. In part due to exclusively focusing on the coverage criteria, traditional test generation tools—such as Evosuite and Randoop—have been critiqued for their lack of readability and understandability of tests generated as judged by practitioners [16, 12] and have limited adoption in the industry [4].

2.2 Large Language Models

Large Language Models (LLMs) are generative language models that are built using deep learning techniques (often employing a Transformer-based [30] architecture). LLMs can be pre-trained (trained) on massive corpora of text and source code, and can then be fine-tuned (trained further) for specific tasks. LLMs can often be controlled via a prompt. The generations of a LLM can be affected using carefully constructed prompts (prompt engineering) to attain particular outcomes [21].

LLMs have exhibited promising results in a range of software engineering tasks, including programming language translation [31], code completion [13], and code summarization [1]. Notably, LLMs demonstrate a remarkable capacity to generate completions that appear natural and coherent in both natural language and source code contexts [26]. In part due to the naturalness of LLM generations—which is lacking in the output of traditional test generation tools—researchers have started to train or use existing LLMs for generating unit tests [26, 29, 5, 28]. Research on test generation using LLMs has demonstrated the ability of LLMs to produce readable unit tests [29, 33].

2.3 GitHub Copilot

GitHub Copilot is a tool developed by GitHub[1] and OpenAI[2] to make code suggestions (generations) directly within well-known IDEs. Copilot's generations have been shown to improve perceived developer productivity in a study from GitHub itself [34]. Furthermore, several studies have demonstrated that Copilot's generations have a similar readability and understandability to human-written code [2, 22]. Copilot employs a distinct version of OpenAI's Codex LLM to create it generations [9]. Codex has been fine-tuned on publicly available open-source GitHub projects [9]. A key difference between using Codex and Copilot for code generations is that the prompt used to attain generations from the LLM can *not* be dictated when using Copilot.

2.4 Large Language Models for Test Generation

We discuss several papers that use LLMs for unit test generation and how they relate to our work.

Lahiri et al. [20] develops TiCoder, a system that addresses the problem of the semantic gap between the informal natural language used to produce a generation (i.e., the user specified prompt) and the actual code generation. They advocate for "test-driven user-intent discovery" [20], where direct user feedback is used to "refine and formalize the user intent through generated tests." The tests are generated using OpenAI's Codex LLM [9]. In the proposed system, TiCoder, the focus is on establishing the correct user intent and then generating tests that correspond to the user intent. We are particularly concerned with the usability of test generated of already existing code, instead of test generations which are created to conform to new code generations.

Tufano et al. [29] develops AthenaTest, an LLM-based approach which is capable of generating unit tests for Java. They pre-train an LLM using natural language and Java source code, and then fine-tune it using Java unit tests and their focal methods (i.e., the method(s) being exercised by the respective unit tests) [29]. They compare AthenaTest tests generations to tests generated by EvoSuite [14] and OpenAI's GPT-3 LLM [8]. They find that AthenaTest "achieves comparable or better test coverage" while at the same time being more readable, understandable, and effective as judged by professional practitioners [29]. Lemieux et al. [21] develops CodaMosa, a technique combining search-based software testing (SBST) with Codex to achieve a higher structurual coverage. CodaMosa uses Mosa [25], a mutation-based test generation approach, until no more additional coverage can be achieved; then it uses Codex [9] to generate a test for methods which still have

[1]GitHub: `https://github.com/`

[2]OpenAI: `https://openai.com/`

a low coverage. They evaluate CODAMOSA on over 400 open-source Python modules and find that CODAMOSA achieves “statistically significantly higher coverage” on most modules when compared to using only SBST or Codex for test generation. Unlike Tufano et al. [29] and Lemieux et al. [21] we do not develop a tool or train a LLM to generate tests for a particular programming language with the goal of maximizing or increasing coverage. Instead, we focus on investigating how usable test generations from GitHub Copilot are, within an IDE for different use cases (e.g., without existing tests or using a certain method comment strategy). Hence, we take a developer-centric perspective, and investigate how Copilot *can* or *should* be used.

Schäfer et al. [26] develops an experimental tool named TESTPILOT[3] that is able to generate unit tests for JavaScript projects [26]. The TestPilot tool uses an “off-the-shelf” version of the Codex LLM and employs prompt refiners to generate a test, where the prompt is automatically refined based on whether the previous prompt generated a passing test. For example, if a prompt resulted in a test containing a syntax error, Codex is re-prompted and instructed to fix the error in the generated test. They evaluate the tool on 25 open-source JavaScript projects. TestPilot achieves a median statement coverage of 68.2% and the majority of generated tests included “at least one assertion that exercises functionality from the package under test” [26]. A key difference between the work presented in this thesis and the TESTPILOT approach, is that we interact with GitHub Copilot rather than using OpenAI’s Codex model. As a result, we are unable to directly influence or view the prompt which is used to prompt the LLM underlying GitHub Copilot when making a generation. This difference does not only exist for the approach of Schäfer et al. [26], but also for all other LLM test generation approaches discussed in this section.

Siddiq et al. [28] investigates the Java test generation ability of the Codex [9], CodeGen [23], and OpenAI’s GPT-3.5 [10] LLM. They construct varying prompt scenarios. For example, containing the full code of the class under test, with code comments and relevant imports needed for unit testing in Java. They find that across all models, numerous generated test were not compilable, even after they employed rule-based repairs to fix these generated test [28]. 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] investigates the Java test generating ability of Codex [9], 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. [28] and Bareiß et al. [5] we investigate the test generation ability of a LLM. Among other differences, we use GitHub Copilot, and they directly employ the Codex LLM for their test generations. Furthermore, Siddiq et al. [28] investigates test smells in generations to assess their quality, whereas we consider usability aspects of generations. Another difference is that we evaluate every generation as is, whereas Siddiq et al. [28] attempts to apply rule-based repairs and then evaluates the generation. A key difference between our work and the work of Bareiß et al. [5] is that we only consider the first

[3]TESTPILOT: `https://githubnext.com/projects/testpilot/`

generation provided by Copilot, whereas they generate 100 test candidates for each method under test so that it can be compared against Randoop.

Xie et al. [32] develops CHATUNITEST, a ChatGPT-based[4] approach for automatically generating unit tests. The approach generates a ChatGPT prompt with "adaptive focal context" (referring to the code under test) to generate tests. If needed, it uses rule-based repair, and prompt refining (similar to TESTPILOT [26]) to repair generations that are broken tests. CHATUNITEST achieves similar coverage to EvoSuite on the projects which they evaluated, and a substantially higher (method) coverage than ATHENATEST [29]. Relatedly, Yuan et al. [33] investigates the test generation ability of ChatGPT, and develops CHATTESTER an approach that uses prompt refining to repair broken tests. The initial prompt includes the focal method (code under test), 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. Their proposed approach, CHATTESTER, uses prompt refining to repair broken tests which yields a substantial improvement over using ChatGPT directly.

Both Xie et al. [32] and Yuan et al. [33] use ChatGPT for their test generation. Their approaches are similar to each other. They both also differ from the work presented in this thesis in at least two ways: (1) they employ ChatGPT instead of GitHub Copilot (which directly integrates in IDE) for test generations (2) they both use a form of prompt refining, whereas we only consider the generation outputted by GitHub Copilot. Nonetheless, the usability aspects proposed in this thesis are partially based on the test quality factors used to evaluate CHATUNITEST [32], but instead of considering compile correctness, we look at runtime correctness (among other differences).

[4]OpenAI's ChatGPT3.5: `https://platform.openai.com/docs/guides/gpt/managing-tokens`

Chapter 3

Study Design

The goal of our study is to understand how GitHub Copilot performs with regard to test generation. We randomly sample tests from open-source projects and use Copilot to "re-generate" the tests in different scenarios (as specified by the research questions). Within this chapter we discuss how we select the projects and sample tests from those projects, we devise usability aspects to investigate generations, and communicate how we have used Copilot to invoke generations.

3.1 Project Selection

In order to answer the research questions we study the usability of Copilot's test generation ability using seven open-source Python projects. We chose Python projects as Python is one of the best supported programming languages in the Codex model (Copilot is powered by a Codex model) [9].

Project	Domain	Provider	Stars	LOC	# Test Classes	# Tests Methods
click	CLI	GitHub	13941	21371	0	327
pyexperiment	research	GitHub	181	6492	36	239
django-multiurl	web development	GitHub	274	266	1	8
python-crontab	DevOps	GitLab	83	1927	19	176
exif	image handling	GitLab	29	6007	7	51
python-lottie	file manipulation	GitLab	112	24307	35	195
pyspread	GUI application	GitLab	47	21481	14	173

Table 3.1: List of open-source Python projects from which tests were sampled.

Projects were selected from GitLab[1] and GitHub. Codex has been trained on public GitHub open-source projects [9]. 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

[1]GitLab: `https://about.gitlab.com/`

we conjecture that the Codex LLM underlying Copilot has been trained on source code from more popular open-source projects. We find that similar LLMs found in the literature often use popular open-source projects for their pre-training [13, 31]. 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[2] or unittest[3] framework, to simplify the coverage analysis needed in this study. The final set of projects in Table 3.1 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.

3.2 Sampling and 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 (i is the identifier of a test) 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 or original test T_i. For example, a generated test might be a failing generation 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.

3.3 Aspects of Usability

We created the usability aspects to gauge the usability of (Python) test generations from Copilot. In general, we define a usable generation as a test generation from Copilot that could be directly used in a test suite without any modification. In particular, we define and justify our set of aspects that represent usability 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.

[2]pytest: `https://docs.pytest.org/en/7.2.x/`

[3]unittest: `https://docs.python.org/3/library/unittest.html`

Runtime Correctness A generation without syntax errors may still give runtime errors from the Python interpreter. For example, this occurs when a generation G_i is passing an incorrect value to a parameter of a method. Similar to generations with syntax errors, 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 assertion oracles in the generated test are correct, but the code under test is flawed. Similarly, a passing test might unintentionally comply with faulty behavior exhibited by the code under test. We assume that because the code under test is tested by the original test T_i, that the code under test is correct. Hence, a generation that is a failing test requires modification before it can be employed in a test suite, which negatively impacts the usability.

Non-Triviality Passing tests may contain no assertions or have tautological assertions. As a result the tests appears to be passing but in reality no meaningful behavior is being asserted. This would thus require modification of the generated test, as the test oracles would need to be added to check the correctness of the code under test, which in turn negatively impacts the usability.

Coverage When considering a test pair (T_i, G_i) we can determine the branches covered of 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. G_i covering fewer of the same branches indicates that fewer branches are exercised than the human-written test T_i. In turn, the generation G_i would require modification to ensure it fully 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, a less *suitable* test, and hence negatively impacting the usability.

Syntactic Correctness, Runtime Correctness, Passing, and Non-Triviality are determined for all Copilot generations using the iteratively assigned code aspects labels. It should be noted that a generation G_i can have multiple code aspects labels (e.g., passing and trivial).

3.4 Invoking Generations

We invoke a generation from Copilot for a given original test by stripping the test method body, and then “regenerating” the test method body using Copilot directly in an IDE.[4] For example, in Listing 3.1 we have a test method from the `pyspread` project. The test method consists of a method signature (row 1), method comment (row 2), and method body (row

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

3-5). In Listing 3.2 we have the same test method but stripped, the method body has been removed only leaving the method signature and method comment. 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.

```python
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 3.1: Example method from the `pyspread` project.

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

Listing 3.2: Stripped example method from the `pyspread` project.

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

3.4.1 With- and Without-Context

We are interested in the usability of the generations when there is an existing test suite (With-Context) and when there is no existing test suite (Without-Context). Thus, for every test invoked within an existing test suite, we also invoke the test without an existing test. Meaning that all test files (except the test file of the original test) are deleted from the test suite. 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 and utility functions are kept.

3.5 Varying Test Method Comments

During the stripping process, the test method comment can be changed. A different method comment can give a different generation. 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: `click`, `django-multirurl`, `python-crontab`, `exif`, and `pyspread`. 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 these five aforementioned projects to simplify the method comment formulations process. Furthermore, we only selected failing test pairs because we assume that practitioners employing Copilot would first directly invoke Copilot to make a generation for a test (due to the ease of invoking a generation) and if that generated test is failing would then attempt to invoke Copilot to generate a more usable test by providing a (instructive) comment.

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 3.3.

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

Listing 3.3: Example method from the `django-multirurl` project.

(1) **Minimal Method Comment**
This type of comments provides a minimal description of a particular test method. An example would be:

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

```

Listing 3.4: 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.” An example would be:

```python
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 3.5: Test method with a Behavior-Driven Development Comment.

(3) **Usage Example Comment**
This type of comments provides a code snippet of a possible call of the code under test. In the usage example we include an explicit usage example and what the example “gives” (if applicable). An example would be:

```python
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 3.6: Test method with a Usage Example Comment.

The example usage does not need to relate to a specific scenario a test is testing, but only to the code under test. This approach is based on earlier LLM test generation work by Max et al. [26].

(4) **Combined**
Finally, we combine all aforementiond comment strategies in one method comment. An example would be:

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

    Given that I resolve a URL
    when that URL does not match
    then an exception should be raised

    example usage:

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

    gives:

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

Listing 3.7: Test method with a Combined Comment.

The method comments were manually formulated by two contributors of this research project. 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.

3.6 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.[5] Suggestions were manually invoked in the PyCharm 2022.2.4 (Professional Edition) 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.[6]

[5]Getting started with GitHub Copilot (Seeing alternative suggestions): `https://docs.github.com/en/copilot/getting-started-with-github-copilot#seeing-alternative-suggestions`

[6]Coverage.py: `https://github.com/nedbat/coveragepy`

Chapter 4

Results

Within this chapter we report the results of RQ_1 and RQ_2, which we combine in one section (which we refer to as $RQ_{1,2}$) so we can discuss each usability aspect With- and Without-Context. We similarly discuss and combine the findings of RQ_3 and RQ_4.

4.1 $RQ_{1,2}$: How useable 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, these test pair form the set O. All considered test pairs $(T_i, G_i) \in O$ were labelled with one or multiple code aspects. All labels assigned (with their definition) can be found in Table 4.2, we created 8 code aspects in total. The code aspect labels were iteratively created, we stopped sampling sets of tests (batches) when there were no more new code aspect labels that could be created (until theoretical saturation).

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 or broken test, or an empty generation. Furthermore, we define a broken test as a test with either a syntax or runtime error. In Table 4.1 a breakdown of all tests generated is provided, where each generated test belongs to

	With-Context $(n = 53)$	**Without-Context** $(n = 53)$
Passing Tests	24 (45.28%)	4 (7.55%)
– Trivial	0 (0.00%)	0 (0.00%)
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 4.1: Breakdown of all Copilot generations for $RQ_{1,2}$ With- and Without-Context.

Code Aspect Name/Label	**Definition**
Assert Mismatch	The generated test contains an assertion that evaluates to false
Empty Generation	An empty generation was received from GitHub Copilot
Incorrect Parameters	The generated test is using the 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	The generated test is using an attribute of an object, but the attribute does not exist or is not subscriptable
Unresolved Reference	The generated test contains a reference to an object which does not exist in the namespace
Failure to Catch Exception	The generated test raises an exception which is not captured, but should be captured (as can be determined from the original test)
Lookup Error	The generated test is using a key of an object, but the key does not exist

Table 4.2: List of code aspect labels with their definition.

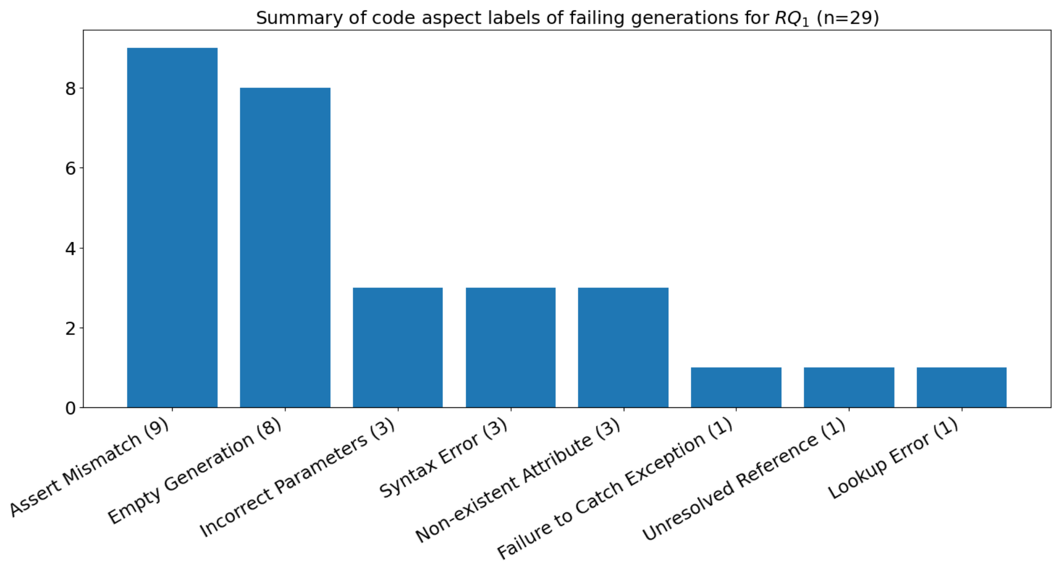

Figure 4.1: Summary of code aspects of failing generations With-Context (RQ_1).

one category based on their code aspect labels. In Figure 4.1 and Figure 4.2 we can find a summary of code aspects of failing generations for With- and Without-Context respectively. We discuss the usability aspects of these generations in the following subsections.

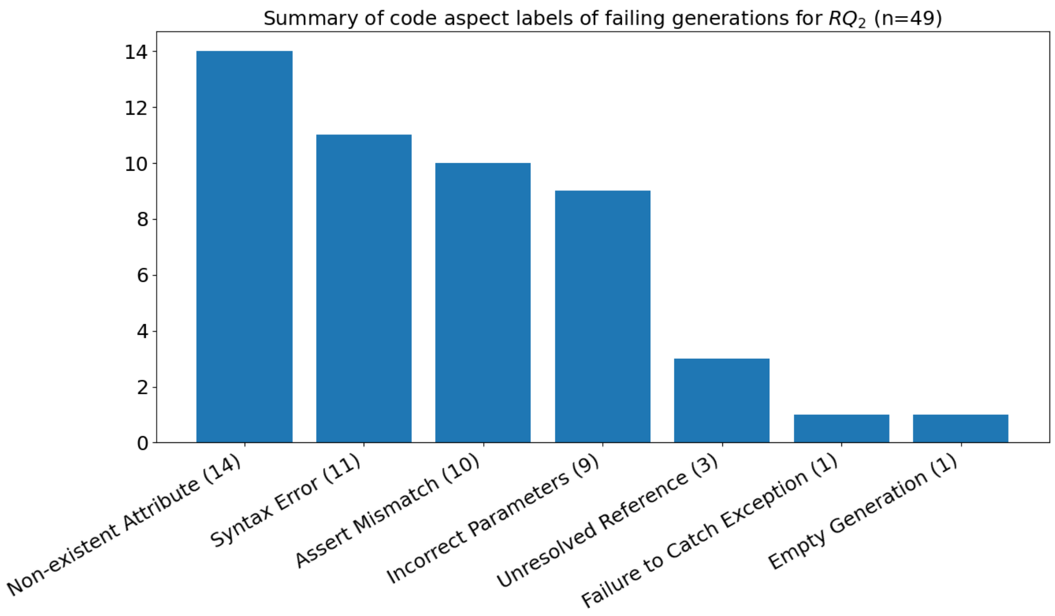

Figure 4.2: Summary of code aspects of failing generations Without-Context (RQ_2).

4.1.1 Syntax and Runtime Correctness

In Table 4.1 we find that 5.66% and 16.98% of all generations fail due to a syntax error or runtime error (respectively) With-Context. Without the context of a test suite, 20.75% and 50.94% of all generations fail due to a syntax or runtime error (respectively). We have identified six reasons demonstrating why these syntax and runtime errors occur, and we illustrate them with an example:

(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. For example, the generated test method `test_captures_stdout_stderr` (see Listing 4.1) fails because the last row of the generation is seemingly incomplete—it misses a single closing quote and parenthesis.

```
def test_captures_stdout_stderr(self):
    """Test capturing stdout and stderr from print
    """
    message = "This should be captured..."

    buf = io.StringIO()
    with stdout_err_redirector(buf):
        print(message)
        print(message, file=sys.stderr)

```

```
11      self.assertEqual(buf.getvalue(), message + '\n' + message + '\
```

Listing 4.1: Failing generation due to an Syntax Error. On row 11 the single quote and parenthesis should be closed.

(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, the generated test method `test_option_optional` is using a keyword argument `optional` in the creation of a class instance of an `Option` object (see Listing 4.2). 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 results in failing generation 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 4.2: 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, the generated test method `test_modify_ascii_same_len` (see Listing 4.3) fails because the object `self.image` does not contain the attribute `_update_exif()`.

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

Listing 4.3: 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 4.4) 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.

```python
def test_18_range_cmp(self):
    """Compare ranges"""
    self.assertEqual(CronRange('*/6'), CronRange('*/6'))
    self.assertNotEqual(CronRange('*/6'), CronRange('*/7'))
    self.assertNotEqual(CronRange('*/6'), CronRange('*/6-7'))
```

Listing 4.4: 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. For example, the generated test method `test_cli` (see Listing 4.5) fails because a `SystemExit` exception is thrown by `parser.parse_args()` in some cases (this is a parametrized test).

```python
def test_cli(argv, res):
    """Test cli"""

    with patch.object(sys, 'argv', argv):
        parser = PyspreadArgumentParser()
        args = parser.parse_args()

        if res is not None:
            assert args == res
        else:
            assert args is None

```

Listing 4.5: Failing generations due to an Failure to Catch Exception. The `parser.parse_args()` can raise an exception which should the be catched.

(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 4.6) fails because the key value `CRON_VAR` does not exist in `self.crontab.env`. Another form of the Lookup Error is when an out of range index value is used on a list or array.

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

```

Listing 4.6: Failing generations due to an Lookup Error. The key value `self.crontab.env['CRON_VAR']` does not exist.

Comparison With- and Without-Context Within Figure 4.1 and Figure 4.2 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.

4.1.2 Passing and Non-Trivial

In Table 4.1 we find that 45.28% and 7.55% of the generated tests were passing for With- and Without-Context (respectively). All passing tests were non-trivial, and thus contained at least one non-tautological assertion. Our manual inspection of passing tests With-Context reveals that some passing tests appear to "mimic" tests in its direct test context (in the same test file). To demonstrate the mimicking behavior we show a generated test in Listing 4.7 and a test in its direct context in Listing 4.8.

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

Listing 4.7: A test generated by Copilot.

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

Listing 4.8: A highly similiar test in the direct test context of Listing 4.7.

The generated test in Listing 4.7 appears to be mimicking the test in Listing 4.8. 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:[1] $s(T,G_i) = 1 - \frac{d(T,G_i)}{\max(|T|,|G_i|)}$ 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 4.3 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 With-Context, which are even more similar. This implies that Copilot is producing generations which are similar to already existing tests in the direct context. Hence, GitHub Copilot appears to be mimicking tests in the direct context for its test generations. Without-Context, Copilot cannot mimic tests in the direct context (as this has been removed), and hence the generations are less similar.

[1]This approach is based on the work of Schäfer et al. [26] and Ippolito et al. [26] who employed edit similarity (or alike) to determine how similar a test was to a generated test in the context of LLM memorization.

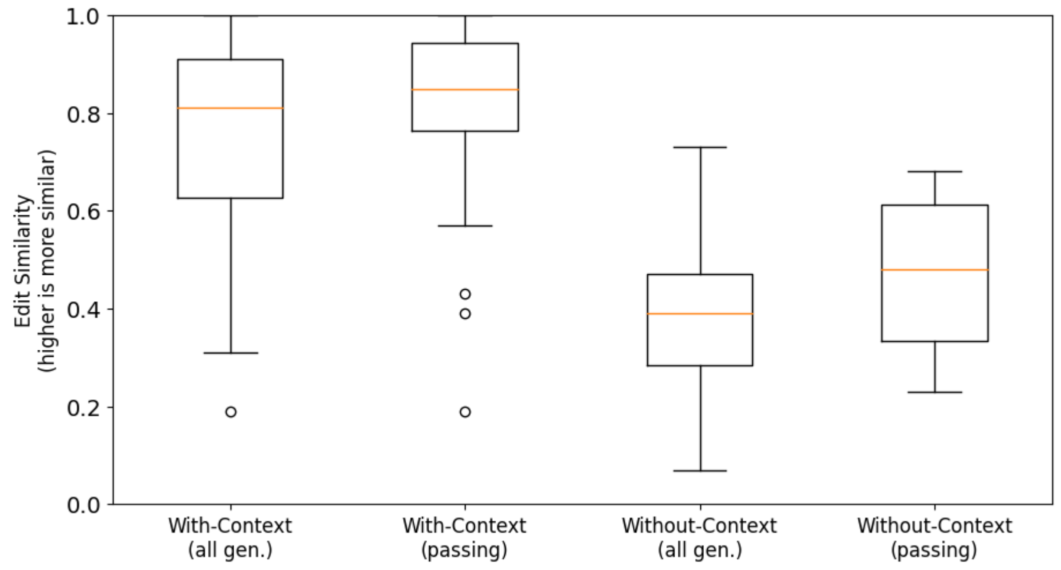

Figure 4.3: Box plot of the edit similarities between every G_i and the most similar test T in the direct context of G_i for $RQ_{1,2}$. An edit similarity of $s(x,y) = 1$ would indicate an exact copy. Empty generations are excluded.

Moving on, we further note that in Table 4.1 we find 16.98% of the tests generated With-Context has no syntax or runtime error, but contain an assertion that evaluates to false (a failing test). In addition, 15.09% of tests generated With-Context are empty. That is, Copilot did not return a generation. In total, combining these two percentages, 32.07% of tests generated With-Context did not contain a syntax or runtime error but did not produce a passing test. We find a similar percentage of failing tests for Without-Context generations, but a 87.5% decrease in Empty Generations. We have identified two reasons demonstrating why failing tests or empty generations occur, we illustrate them with an example:

(1) **Assert Mismatch**
Copilot is not able to determine the expected value of one or multiple assertions for the code under test. For example the generated test method `test_handle_bad_attribute` 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 4.9), 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. Furthermore, the code context did not reveal which string should be matched on.

```
def test_handle_bad_attribute():
    """Verify that accessing a nonexistent attribute raises an
    AttributeError."""
    with open(
4         os.path.join(os.path.dirname(__file__), "grand_canyon.jpg"),
       "rb"
5     ) as image_file:
6         image = Image(image_file)
7
8     with pytest.raises(AttributeError, match="image does not have
      attribute"):
9         image.fake_attribute
```
Listing 4.9: Failing generation due to an Assert Mismatch. The string being matched on (row 8) should be: `"unknown image attribute fake_attribute"`.

(2) **Empty Generation**
Empty Generations[2] occur when Copilot returns no generation. The underlying reason for returning an Empty Generation can only be speculated as the Copilot system is closed-source. However, it could be caused by context length limit of the Codex LLM underlying Copilot [28]. The prompt being formulated by Copilot based on the code context may be too long for the model.

Comparison With- and Without-Context Within Figure 4.1 and Figure 4.2 we note that tests generated Without-Context have a decrease of 87.5% in Empty Generations occurrences when compared to With-Context.

4.1.3 Coverage

We compute the branch coverage[3] for the original test T_i and generated test G_i of every passing test pair $(T_i, G_i) \in O$ for both With- and Without-Context. In total, there are 24 passing tests With-Context, and four Without-Context. In Table 4.3 we find an overview of all passing tests generated With-Context and their coverage data. Recall that for the coverage usability aspect we consider generations G_i covering fewer branches than the original test T_i as negatively impacting the usability. We consider these generations as less suitable with respect to the human-written test T_i; this suitability is captured by the Branch Overlap Ratio in Table 4.3.

When considering passing tests With-Context, we find that 17 of the 24 (#2-18) generated passing tests cover the same branches as their human-written counterpart T_i. Only one test generated (#1) covers the same and more new branches. The remaining five generated tests (#19-24) cover strictly fewer branches and/or cover new branches. Hence, most passing tests generated by GitHub Copilot With-Context do not cover fewer branches than the original test T_i, which positively impacts the usability.

Identical branches being exercised in the test cases (#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

[2] An Empty Generation gives a passing test as the generated test method does not fail. We however count it as a failing generation.

[3] The line coverage was initially also considered but yielded similar results.

#	Test Name	Diff. Covered Branches (T_i, Gi)	Branch Overlap Ratio	New Branches Covered by G_i
1	test_option_custom_class_reusable	+7.38%	1.0	15
2	test_show_true_default_boolean_flag_value	0.0%	1.0	0
3	test_resolve_match_first	0.0%	1.0	0
4	test_resolve_match_last	0.0%	1.0	0
5	test_resolve_match_middle	0.0%	1.0	0
6	test_resolve_match_path_brand	0.0%	1.0	0
7	test_list_all	0.0%	1.0	0
8	test_main_no_processes_long	0.0%	1.0	0
9	test_data_access	0.0%	1.0	0
10	test_on_markup_renderer_pressed	0.0%	1.0	0
11	test_rgb2qimage	0.0%	1.0	0
12	test_insertTable	0.0%	1.0	0
13	test_row	0.0%	1.0	0
14	test_rgb666	0.0%	1.0	0
15	test_set_row_height	0.0%	1.0	0
16	test_get_absolute_access_string	0.0%	1.0	0
17	test_04_number	0.0%	1.0	0
18	test_find_list	0.0%	1.0	0
19	test_06_clear	0.0%	0.98	2
20	test_21_slice_special	0.0%	0.8	13
21	test_progressbar_item_show_func	-1.5%	0.98	0
22	test_remove_layer	-11.36%	0.89	0
23	test_09_removal_during_iter	-16.9%	0.81	2
24	test_command	-19.85%	0.85	0

Table 4.3: Overview of coverage data for all passing test pairs generated With-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 4.3 cover the exact same branches as the original test T_i.

In Figure 4.4 we have an overview of the coverage data of all passing tests Without-Context. We find that only one generated test (#1) covers the same branches as their human-written counterpart. The remaining generated tests (#2-4) cover fewer and/or some new branches. This indicates that tests generated Without-Context, even if passing, are less suitable.

#	Test Name	Diff. Covered Branches (T_i, Gi)	Branch Overlap Ratio	New Branches Covered by G_i
1	test_get_absolute_access_string	0.0%	1.0	0
2	test_09_removal_during_iter	-4.23%	0.78	14
3	test_get_context_objects_missing	-15.89%	0.87	1
4	test_handle_bad_attribute	-95.92%	0.06	0

Table 4.4: Overview of coverage data for all passing test pairs generated Without-Context.

4.2 $RQ_{3,4}$: 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?

We consider test pairs from O who belong to a subset of the projects considered in this investigation: `click`, `django-multiurl`, `python-crontab`, `exif`, and `pyspread`. Furthermore, we only consider failing test pairs (see Section 3.5 for details). In total, we select 23 tests T_i for which the respective generation G_i 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 in particular interested which of these four method comment strategies result in the most usable test generations. Hence, we compare each method comment strategy to each other 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 result 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 4.2.

In Table 4.5 we find a breakdown of all test generations for each method comment strategy, both With- and Without-Context. This breakdown is constructed similarly to the breakdown in Section 4.1. Figure 4.4 shows a summary of code aspects of failing generations for both With- and Without-Context for all considered method comment strategies. We compare the usability aspects of the generations for each method comment strategy in the following subsections.

4.2.1 Syntax and Runtime Correctness

In Table 4.5 we find that With-Context the **Usage Example Comment** yields the least number of tests with syntax or runtime errors (broken tests). Without-Context the **Combined Comment** strategy yields the least number of broken tests. In Figure 4.4 we find which type of runtime errors occur for all the different method comment strategies, which reveals

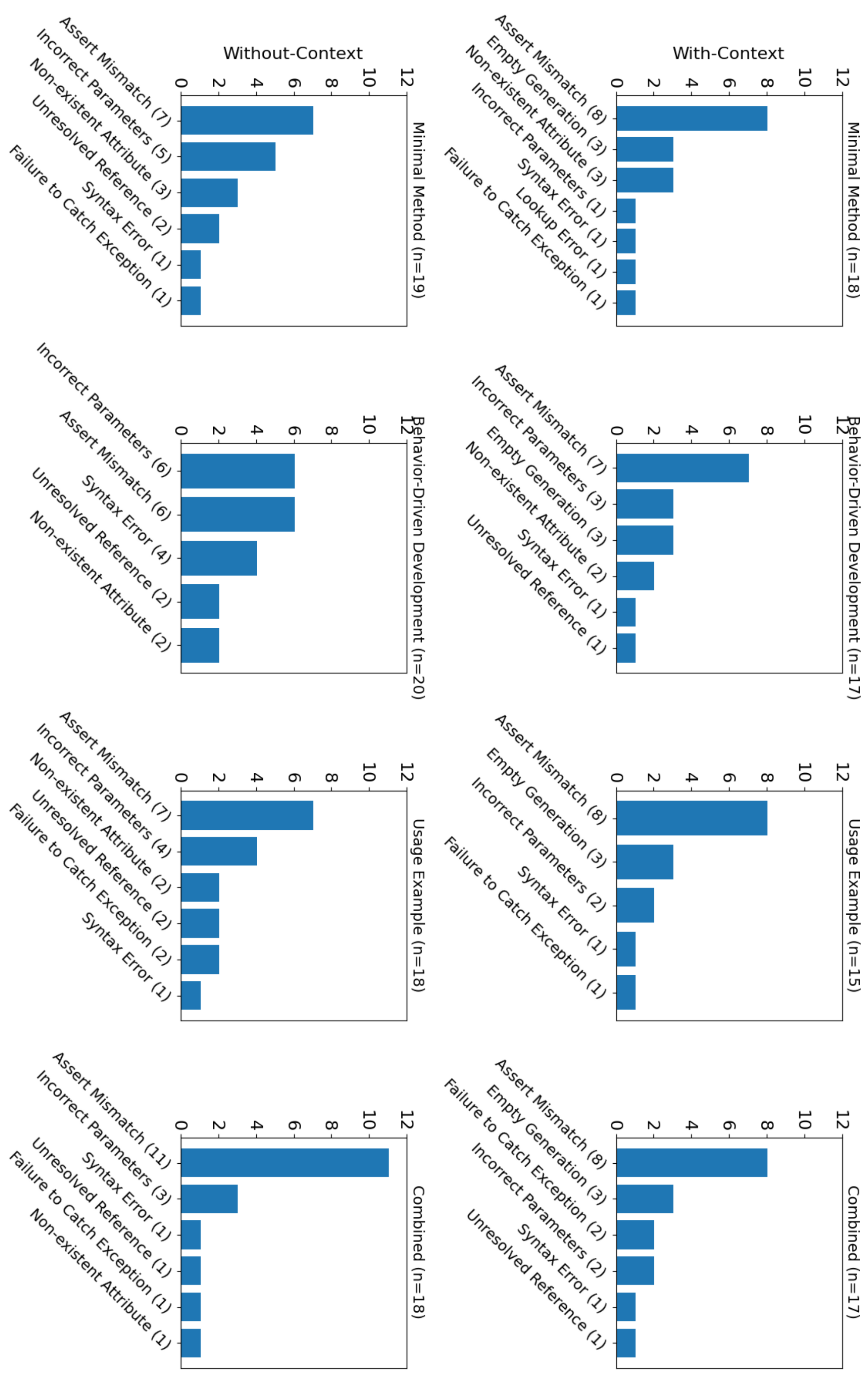

Figure 4.4: Summary of code aspects of failing generations for each method comment strategy for both With- and Without-Context ($RQ_{3,4}$).

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%)
– Trivial	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
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%)**
– Trivial	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
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 4.5: 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.

why the generations fail. We find the same reasons as found in $RQ_{1,2}$ apply here but with different distributions for every method comment strategy.

Comparison With- and Without-Context In Figure 4.4 we note that, independent of the comment strategy applied, a Without-Context generation results in an increase of broken tests. Notably, the overall occurrence of Incorrect Parameters and Unresolved References increases for generations Without-Context.

4.2.2 Passing and Non-Trivial

In Table 4.5 we find that all passing tests are non-trivial, independent of the method comment strategy applied. With-Context the **Usage Example Comment** yields the most passing test. Without-Context, both the **Usage Example Comment** and **Combined Comment** produce the same number of passing tests. Furthermore, in Figure 4.5 we similarly find that GitHub Copilot is mimicking existing tests in the direct test context (see Section 4.1.2), although the effect is less pronounced.

Comparison With- and Without-Context We observe that in Figure 4.4 generations Without-Context produce no Empty Generations independent of method comment strategy,

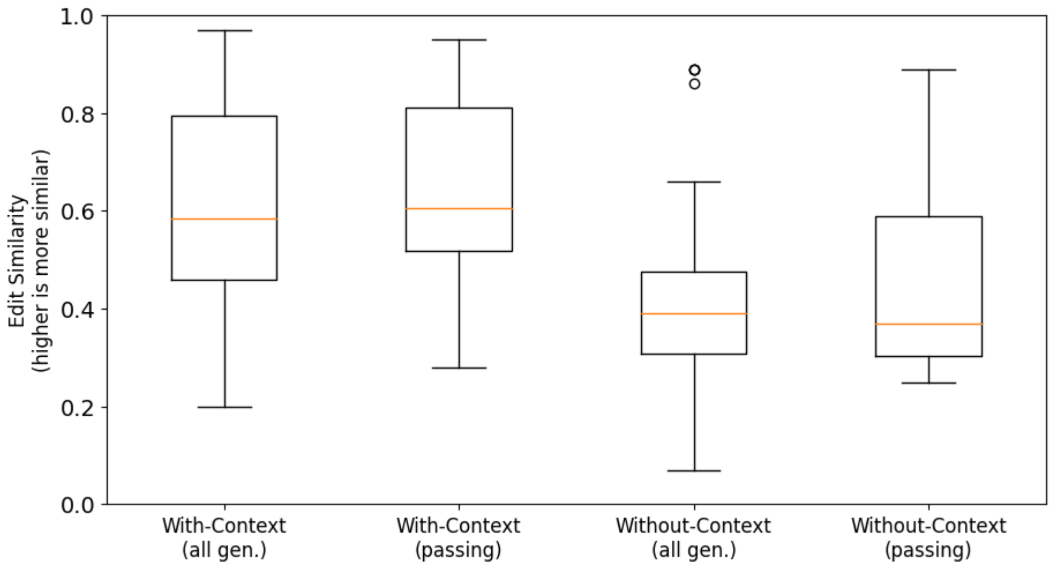

Figure 4.5: Box plot of the edit similarities between every $G_{i,k}$ and the most similar test T in the direct context of $G_{i,k}$ for $RQ_{3,4}$. An edit similarity of $s(x,y) = 1$ would indicate an exact copy. Empty generations are excluded.

this is in similar to our earlier finding in $RQ_{1,2}$ regarding Empty Generations. Namely, generations made Without-Context receive fewer Empty Generations from Copilot, possibly due to the limited context length of the underlying model of Copilot.

4.2.3 Coverage

We compute the branch coverage 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, both With- and Without-Context. In Table 4.7 and Table 4.8 we find an overview of all passing tests and their coverage data for With- and Without-Context (respectively). Recall that for the coverage usability aspect we consider covering fewer of the same branches as the original test T_i as negatively impacting the usability, and that we are interested in the method comment strategy that results in the most usable test generations. 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 Branch Overlap Ratio (BOR) over all test generated. Concretely, we compute the average BOR over all test generations[4] considered in $RQ_{3,4}$ ($n = 23$) for each method comment strategy.

In Table 4.6 we find an overview of the average BOR. We observe that the **Usage Example Comment** yields the highest average BOR when considering all test generated With-Context when compared to other strategies. Likewise, we find that the **Combined Com-**

[4]Generations which are failing have a BOR of 0 by definition. Passing tests where the BOR is Not Available (NA) are considered to have a BOR of 1.

	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 (all generated tests)	0.21	0.25	**0.34**	0.26	0.17	0.13	0.2	**0.21**

Table 4.6: 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.

ment strategy produces test generations with the highest average BOR when compared to other strategies for Without-Context generations.

#	Method Comment Strategy	Test Name	Diff. Covered Branches $(T_i, G_{i,k})$	Branch Overlap Ratio	New Branches Covered by $G_{i,k}$
1	UEC	test_handle_bad_attribute	+102.04%	1.0	56
2	UEC	test_18_range_cmp	+0.97%	1.0	1
3	UEC	test_reverse	0.0%	NA	0
4	UEC	test_07_reboot	0.0%	1.0	0
5	UEC	test_get_context_objects_missing	0.0%	1.0	0
6	UEC	test_selection_blocks	0.0%	1.0	0
7	UEC	test_confirm_repeat	0.0%	0.98	4
8	UEC	test_option_optional	0.0%	0.88	19
9	MMC	test_07_reboot	+0.98%	0.98	3
10	MMC	test_group_group_class	0.0%	1.0	0
11	MMC	test_get_context_objects_missing	0.0%	1.0	0
12	MMC	test_mono	0.0%	1.0	0
13	MMC	test_option_optional	-8.87%	0.86	12
14	CC	test_reverse	0.0%	NA	0
15	CC	test_no_match	0.0%	1.0	0
16	CC	test_07_reboot	0.0%	1.0	0
17	CC	test_get_context_objects_missing	0.0%	1.0	0
18	CC	test_selection_blocks	0.0%	1.0	0
19	CC	test_mono	0.0%	1.0	0
20	BDDC	test_handle_bad_attribute	0.0%	1.0	0
21	BDDC	test_cli	0.0%	1.0	0
22	BDDC	test_mono	0.0%	1.0	0
23	BDDC	test_get_context_objects_missing	-0.93%	0.99	1
24	BDDC	test_make_pass_decorator_args	-2.63%	0.9	12
25	BDDC	test_18_range_cmp	-3.88%	0.94	2

Table 4.7: Overview of coverage data for all passing test pairs in M generated With-Context. NA indicates that the test T_i did not cover any branches of the code under test (e.g., the unit test is testing a method of a third-party library).

#	Method Comment Strategy	Test Name	Diff. Covered Branches $(T_i, G_{i,k})$	Branch Overlap Ratio	New Branches Covered by $G_{i,k}$
1	UEC	test_07_reboot	+0.98%	0.98	3
2	UEC	test_reverse	0.0%	NA	0
3	UEC	test_18_range_cmp	-1.94%	0.97	1
4	UEC	test_make_pass_decorator_args	-17.54%	0.83	3
5	UEC	test_06_env_access	-18.09%	0.8	2
6	MMC	test_group_group_class	+136.36%	1.0	78
7	MMC	test_18_range_cmp	+6.8%	0.97	10
8	MMC	test_07_reboot	+0.98%	0.98	3
9	MMC	test_mono	0.0%	1.0	0
10	CC	test_modify_ascii_same_len	+83.33%	1.0	51
11	CC	test_modify_ascii_shorter	+83.33%	1.0	51
12	CC	test_mono	0.0%	1.0	0
13	CC	test_18_range_cmp	-1.94%	0.97	1
14	CC	test_06_env_access	-12.77%	0.85	2
15	BDDC	test_no_match	0.0%	1.0	0
16	BDDC	test_cli	0.0%	1.0	0
17	BDDC	test_mono	0.0%	1.0	0

Table 4.8: Overview of coverage data for all passing test pairs in M generated Without-Context. NA indicates that the test T_i did not cover any branches of the code under test (e.g., the unit test is testing a method of a third-party library).

Chapter 5

Discussion

Within this section we discuss the results and answer each research question. Furthermore, we attempt to formulate implications from our results. Finally, we discuss the threats to validity.

5.1 Usability of GitHub Copilot's Test Generations ($RQ_{1,2}$)

We find that GitHub Copilot is able to generate usable tests when the intended test is similar to an existing test in the test file. Tests generated Without-Context were failing generations in 92.45% of the tests evaluated, this is a stark contrast to tests generated With-Context which only resulted in 54.72% failing generations. Our findings indicate that Copilot heavily depends on the direct test context to generate a passing tests. In addition, we note that generations mimic existing tests in the context, which further strengthens the finding that Copilot depends on the direct context.

Observation I

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.

Interestingly, our findings suggest that Copilot does not appear to consider the code under test. 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 code under test, 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 the Codex LLM which powers Copilot [9, 28].

All projects which are considered in this research separate the test files from the code files. Arguably, if tests are written in the same file as the code under test, Copilot may produce more usable generation due to the presence of the code under test in the context. The code under test could then provide information that would lead a test generation to not contain Incorrect Parameters or using Non-existent Attributes, as the code context can provide examples of the correct usage.

Observation II

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

We find that 55.66% of tests generated (combining both With- and Without-Context) are broken tests or empty generations. These generations are particularly harmful to the usability of Copilot's test generation, as they can require substantial modification before the generated test can be employed.

Failing tests are another culprit of Copilot's test generation ability, with 17.92% of tests generated (combining both With- and Without-Context) being a failing test. To determine correct asserts is a hard problem within itself, which is known as the "test oracle problem" [6]. The test oracle problem refers to the problem of determining correct and incorrect behavior, so that correct asserts can be formulated. Within our usability analysis of generations, 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 code under test is faulty. Likewise, a passing test may be conforming to faulty behavior of code under test. Nonetheless, we exclusively focus on usability of generations, and not the fault-finding ability of generations. Additionally, we assume that the code under test 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 code under test.

We find that 75% of passing tests generated With-Context cover the exact same branches (or more) as their human-written counterparts (the intended test). Without-Context only 25% (1 out of the 4 passing tests) cover the same branches. We consider generations covering the same branches as their human-written counterpart as more suitable than generations who do not. This indicates that passing tests generated by Copilot With-Context do capture the "intention" of the test method signature (and comment, if available) quite well, as the branches executed by the generated test are the same branches executed by the same test written by a human. With-Context generations often covering the same branches as their human-written counterpart can be partially explained by the mimicking behavior of Copilot (see Observation I).

Observation III

Our findings suggest that 75% of all passing tests generated by GitHub Copilot within the context of an existing test suite cover the exact same branches (or more) as the same tests written by humans.

Answer to RQ_1

How usable is a test generated by GitHub Copilot *within* the context of an existing test suite?
Less than half (45.28%) of all generations invoked within the context of an existing test suite are passing tests. The majority (54.72%) of generations are failing, broken, or empty tests. All passing tests contain non-tautological assertions and often cover the same branches as their human-written counterpart. Due to the majority of generations being failing generations, the overall usability of GitHub Copilot's test generation ability is considerably negatively affected, as close to half of all generations will require modification before being used in a test suite. Hence, the usability of tests generated by GitHub Copilot within the context of an existing test suite is poor.

Answer to RQ_2

How usable is a test generated by Copilot *without* the context of an existing test suite?
A minority (7.55%) of all generations invoked without the context of an existing are passing tests. The vast majority (92.45%) of generations are failing, broken, or empty tests. All passing tests contain non-tautological assertions and mostly cover fewer of the same branches as their human-written counterpart. Due to the vast majority of generations being failing generations and passing tests covering fewer branches, the overall usability of GitHub Copilot's test generation ability is severely negatively affected, as almost all generations will require modification before being used in a test suite. Hence, the usability of tests generated by GitHub Copilot without the context of an existing test suite is extremely poor.

5.2 Influencing GitHub Copilot's Test Generations using Test Method Comments ($RQ_{3,4}$)

Method comments are a way to prime GitHub Copilot to produce a different (potentially, more useable) generation. We have compared the usability of Copilot's test generations when applying four different test method comment strategies: **Minimal Method Comment (MMC)**, **Behavior-Driven Development Comment (BDDC)**, **Usage Example Comment (UEC)**, and **Combined Comment (CC)**.

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. Meaning that the application of any one of the aforementioned method comment strategy would have resulted in less failing, and thus more usable generations. We delve into the usability aspects of all method comment strategies to determine which of these method comment strategies resulted in the most usable generations, for both With- and Without-Context.

Firstly, we note that for all method comment strategies, both With- and Without-Context, there are no trivial passing tests. Furthermore, we note 13.04% of With-Context generations, independent of method comment strategy applied, are Empty Generations. Without-Context, there are no Empty Generations. Earlier, in the results of $RQ_{1,2}$, we noted that tests generated Without-Context have a 87.5% decrease in Empty Generations. Both these findings suggests that tests generated Without-Context are less likely to be Empty Generations. This could be caused by context length limit of the Codex LLM underlying Copilot [28]. The prompt being formulated by Copilot based on the code context may be too long for the model.

Observation IV

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

Secondly, we observe that 34.78% of tests generated using the **Usage Example Comment** strategy are passing tests With-Context and 21.74% are passing tests Without-Context, the highest among all method comment strategies. We find 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 for Without-Context generations. 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.

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

Observation V

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

Thirdly, passing tests that use the **Usage Example Comment** and **Combined Comment** strategy produce the highest average ratio of branches overlapping with their human-written counterparts for With- and Without-Context (respectively). Meaning that these strategies produce the most suitable test for their respective context application.

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?
We find that the **Usage Example Comment** strategy produces the most passing tests (34.78%) and the least amount of broken tests (17.39%) when compared to all other method comment strategies considered within the context of an existing test suite. Furthermore, the Usage Example Comment produces test generations with the highest average ratio of branches overlapping with their human-written counterparts. Hence, a test method comment should be formulated as a Usage Example Comment to attain a usable generation within the context of an existing test suite.

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?
We find that the **Combined Comment** strategy produces the most passing tests (21.74%) and the least amount of broken tests (30.43%) when compared to all other method comment strategies considered without the context of an existing test suite. Furthermore, the Combined Comment produces test generations with the highest average ratio of branches overlapping with their human-written counterparts. Hence, a test method comment should be formulated as a Combined Comment to attain a usable generation without the context of an existing test suite.

5.3 Implications

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 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 existing tests (see Answer to RQ_1). In that case, GitHub Copilot tends to mimic other tests for its generations (see Observation I). 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). Although 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 [26, 5]. Generations from GitHub Copilot mimic their surrounding test context for its generations (see Observation I), essentially treating the surrounding test context as test code examples. Without that context, GitHub Copilot is unable to mimic and thus provides far 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 V). 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 plentiful (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 refers to generations (from language generations model such as the LLM which underpins GitHub Copilot) that are "nonsensical or unfaithful to the provided source content" [19]. Hallucinating is a larger challenge within language generation models [19]. Further research should investigate how these hallucinations, stemming from the language generation models, can be mitigated in the domain of test generation.

Implications for GitHub Copilot Our findings suggest that **GitHub Copilot does not consider the code under test when generating tests** (see Observation II). We hypothesize that GitHub Copilot does not send the code under test 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 a Codex model [9]. Thus, we assume that it is not possible to include all code under test 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 code under test, as much as possible, whenever GitHub Copilot is invoked for generating a test method**. Such a system could use techniques such as static analysis to narrow down the relevant code under test based on code imports/references made in the direct test context, or possibly even the file or method name, and then include the code under test as part of the prompt.

5.4 Threats to Validity

Within this section we discuss the primary threats to the validity of the research presented. In particular, we discuss the internal and external threats, and the threats to reproducibility.

5.4.1 Internal Threats

- Within this work we evaluate a commercial code generation tool (GitHub Copilot). We have treated the GitHub Copilot system as a black-box system, and we do not know what exact code context has been included which led to the generations which we investigated in this research.
- The formulations of the different method comments (Minimal, Behavior-Driven Development, and Usage Example) 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 thesis.
- 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 would not impact the results and findings overall as we investigated and labelled 290 generations.

5.4.2 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. Including more projects (which would lead to more generations) could strengthen the confidence of the findings presented. Furthermore, we exclusively focus on the Python programming language, and do not include other programming languages as part of our analysis.
- The findings presented in this research may not generalize to future versions of GitHub Copilot. This can impact the findings presented if the GitHub Copilot system undergoes (major) changes. We include the GitHub Copilot plugin version used in this research in Section 3.6.

5.4.3 Threats to Reproducibility

The GitHub Copilot system, being a commercial system, may change its underlying LLM in the future, resulting in different code generations than those documented and investigated in this research. This in turn may impact the reproducibility of the results. To mitigate this, we include all generations investigated in this study as part of the replication package [17].

Chapter 6

Conclusions and Future Work

Within this section we provide a gist of the findings of our study by summarizing the results and discussion, and provide several potential future research directions.

6.1 Conclusions

Within this thesis we investigated the usability of in total 290 tests generated by GitHub Copilot in several scenarios: with- and without an existing test suite (test code context), 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 substantially less usable than tests generated within an existing test code context, with the overwhelming majority (92.45%) of test generations being failing, 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 counterpart.

Finally, we find that test method comments using a Usage Example Comment produced the most usable test generations when invoking GitHub Copilot within an existing test code context. Without existing test code context, the Combined Comment strategy yielded the most usable test generations.

6.2 Future work

Within this work, we demonstrate the usability of tests generated by Copilot in projects written using the Python programming language. A natural suggestion for future work is to investigate Copilot's output in test suites written in different programming languages and to potentially compare them (as far as it is possible). In particular, statically typed programming languages might produce different reasons for failing generations than the ones we have seen so far.

We have only considered one particular commercial code generation tool: GitHub Copilot. There do also exist other commercial code generation tools (such as Tabnine[1]). Furthermore, there are code generation models in existence in research [13, 31] or in the open-source domain (such as FauxPilot[2]). It would be useful information for software practitioners to know how the test generation usability of all these different tools and models relate. The usability aspects defined within this thesis can be applied to investigate the usability of other code generation tools, as the aspects are tool agnostic.

A newly released blog article[3] from GitHub suggests that newer versions of GitHub Copilot consider other code files which are open in the IDE for its generations. Future research on GitHub Copilot's test generations should consider how opening certain code or test files in the IDE impacts the usability of Copilot's test generations.

[1]Tabnine: `https://www.tabnine.com/`

[2]FauxPilot: `https://github.com/fauxpilot/fauxpilot`

[3]"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/`

Bibliography

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.449. URL `https://aclanthology.org/2020.acl-main.449`.

[2] Naser Al Madi. How readable is model-generated code? examining readability and visual inspection of github copilot. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5, 2022.

[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. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2013.02.061. URL `https://www.sciencedirect.com/science/article/pii/S0164121213000563`.

[4] Andrea Arcuri. An experience report on applying software testing academic results in industry: We need usable automated test generation. *Empirical Softw. Engg.*, 23 (4):1959–1981, aug 2018. ISSN 1382-3256. doi: 10.1007/s10664-017-9570-9. URL `https://doi-org.tudelft.idm.oclc.org/10.1007/s10664-017-9570-9`.

[5] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 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. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

[7] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page

179–190, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786843. URL https://doi.org/10.1145/2786805.2786843.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[9] 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[10] Xuanting Chen, Junjie Ye, Can Zu, Nuo Xu, Rui Zheng, Minlong Peng, Jie Zhou, Tao Gui, Qi Zhang, and Xuanjing Huang. How robust is gpt-3.5 to predecessors? a comprehensive study on language understanding tasks. *arXiv preprint arXiv:2303.00293*, 2023.

[11] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, Nov 2014. doi: 10.1109/ISSRE.2014.11.

[12] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786838. URL https://doi.org/10.1145/2786805.2786838.

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

[14] Gordon Fraser and Andrea Arcuri. 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 '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025179. URL https://doi.org/10.1145/2025113.2025179.

[15] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, 24(4), sep 2015. ISSN 1049-331X. doi: 10.1145/2699688. URL https://doi.org/10.1145/2699688.

[16] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. An empirical investigation on the readability of manual and generated test cases. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 348–3483, 2018.

[17] Khalid El Haji. Empirical Study on Test Generation Using GitHub Copilot — Replication Package, June 2023. URL `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. 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. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[20] Shuvendu K Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950*, 2022.

[21] Caroline Lemieux, Jeevana Priya Inala, Shuvendu Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *ICSE'23*, May 2023. URL `https://www.microsoft.com/en-us/research/publication/codamosa-escaping-coverage-plateaus-in-test-generation-with-pre-trained-large-language-models/`.

[22] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 1–5, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528470. URL `https://doi-org.tudelft.idm.oclc.org/10.1145/3524842.3528470`.

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

[24] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938657. doi: 10.1145/1297846.1297902. URL `https://doi.org/10.1145/1297846.1297902`.

[25] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of

the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018. doi: 10.1109/TSE.2017.2663435.

[26] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527*, 2023.

[27] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. On the effectiveness of manual and automatic unit test generation: Ten years later. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 121–125, 2019.

[28] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Exploring the effectiveness of large language models in generating unit tests. *arXiv preprint arXiv:2305.00418*, 2023.

[29] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. 2020.

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

[31] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 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*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL `https://aclanthology.org/2021.emnlp-main.685`.

[32] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. Chatunitest: a chatgpt-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764*, 2023.

[33] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

[34] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 21–29, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392730. doi: 10.1145/3520312.3534864. URL `https://doi.org/10.1145/3520312.3534864`.