Detecting Test Smells in Python Test Code Generated by LLM: An Empirical Study with GitHub Copilot

Victor Anthony Alves
Federal University of Ceará (UFC)
Quixadá, Ceará, Brazil
victorpa@alu.ufc.br

Cristiano Santos
Federal University of Bahia (UFBA)
Salvador, Bahia, Brazil
cristianossd@gmail.com

Carla Bezerra
Federal University of Ceará (UFC)
Quixadá, Ceará, Brazil
carlailane@ufc.br

Ivan Machado
Federal University of Bahia (UFBA)
Salvador, Bahia, Brazil
ivan.machado@ufba.br

ABSTRACT

Writing unit tests is a time-consuming and labor-intensive development practice. Consequently, various techniques for automatically generating unit tests have been studied. Among them, the use of Large Language Models (LLMs) has recently emerged as a popular approach for automatically generating tests from natural language descriptions. Although many recent studies are dedicated to measuring the ability of LLMs to write valid unit tests, few evaluate the quality of these generated tests. In this context, this study aims to measure the quality of the test codes generated by GitHub Copilot in Python by detecting test smells in the test cases generated. To do this, we used approaches to generating unit tests by LLMs that have already been applied in the literature and collected a sample of 194 unit test cases in 30 Python test files. We then measured them using tools specialized in detecting test smells in Python. Finally, we conducted an evaluation of these test cases with software developers and software quality assurance professionals. Our results indicated that 47.4% of the tests generated by Copilot had at least one test smell detected, with a lack of documentation in the assertions being the most common quality problem. These findings indicate that although GitHub Copilot can generate valid unit tests, quality violations are still frequently found in these codes.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Artificial Intelligence → Large Language Models**; • **General and reference → Empirical studies**.

KEYWORDS

Test Smells, Large Language Models, Python Test Code

1 INTRODUCTION

Unit testing is a foundation of software quality assurance, providing a mechanism for evaluating the functionality of individual units of source code in isolation [11]. Despite its critical role, the creation of unit test code is often overlooked by developers, predominantly due to its complexity and the time-consuming nature of the task [2, 5, 25]. The manual generation of tests is not only labor-intensive but also costly [17]. Consequently, there is a growing emphasis within the development community on developing tools and methodologies that automate the generation of unit tests [28]. This automation aims to alleviate the charge associated with manual test writing and to enhance overall test effectiveness and efficiency.

Recent research by Schäfer et al. [27] and El Haji et al. [7] highlights the substantial capabilities of Large Language Models (LLMs) in generating test code. These studies commonly employ a strategy where LLMs, such as ChatGPT[1], GitHub Copilot[2], and Amazon CodeWhisperer[3], are pre-trained using natural language prompts. Such prompts may drive test code generation that aligns with the anticipated outcomes. This method capitalizes on the models' human-like text comprehension and generation abilities, thereby facilitating the creation of customized test code for distinct use cases [35, 37]. LLMs' depth of knowledge and contextual understanding enables them to transcend traditional testing limitations, probing diverse scenarios and identifying potential issues that conventional methods might overlook.

While LLMs are increasingly indicated as the future of code generation, a growing body of research is focused on evaluating the quality of the code they produce. Hansson and Ellréus [13] have noted that code generated by models such as ChatGPT and GitHub Copilot sometimes violates established quality guidelines. Furthermore, in production code crafted by various LLMs, Yetiştiren et al. [36] has conducted a thorough analysis of 'code smells'—indicators of deeper issues affecting maintainability and reliability. The findings suggest that, despite the correctness of most generated code, there are significant concerns regarding its maintainability and trustworthiness

In the domain of test code generation, although numerous studies have addressed the generation capabilities of LLMs, a significant research gap remains concerning the quality of test code produced by these models [7, 29, 30]. Traditional testing methodologies often encounter issues with test cases, commonly known as *test smells*, which may signal deficiencies in test design or implementation. These shortcomings can lead to decreased efficiency in detecting failures or validating software behavior [31]. Test smells can manifest in various forms, such as poorly structured test code or overly complex logic, which complicates both understanding and maintenance of the test code [16].

[1] https://chat.openai.com/
[2] https://github.com/features/copilot
[3] https://aws.amazon.com/pt/codewhisperer/

In this context, the main objective of this research project is to conduct an empirical study on the quality of test code generated by LLMs, with a specific focus on GitHub Copilot for Python. The aim is to investigate whether the tests generated by these tools can be trusted and whether the code produced is well structured. To achieve this, unit test codes were generated from open source Python projects. After generating the test code, a detailed analysis was carried out to assess its quality. This process involved the use of specialized tools to identify *test smells*. Finally, we carried out an evaluation of the test cases generated by Copilot by software quality and development professionals. This study provides findings on the reliability of test codes generated by GitHub Copilot, focusing on the frequency of detection of test smells, the most common types of test smells and identifiable patterns, as well as practitioners' perceptions of the quality of these codes.

2 BACKGROUND

2.1 Unit Testing & Test Smells

Unit testing is a technique designed to detect defects and validate the functionality of the smallest testable parts of software, such as modules, objects, and classes, which can be examined in isolation [12]. Automated frameworks, such as JUnit[4] for Java, have facilitated the widespread adoption of this method, enabling the frequent and automatic execution of unit test suites [6]. This practice is instrumental in preventing programming errors and identifying issues early in the development cycle [14, 24].

In the context of unit testing, certain problems, known as test smells, may affect the quality of the tests. These issues generally result from poor design choices made during the implementation of test cases, significantly affecting test effectiveness [23, 32]. Test smells often originate when test code is initially committed to a repository and are likely to persist, adversely affecting the software's maintainability and directly impacting its quality [15, 26, 30].

2.2 Automatic test generation

Automated test generation facilitates the production and execution of numerous inputs that thoroughly test software units [34]. Traditional approaches to test generation have included model-driven, requirement-based, static analysis, and research-driven techniques [19, 21].

The effectiveness of automatic test generation tools in identifying software failures has been demonstrated in studies on open-source software [1]. Tools such as Evosuite[5][9] and Randoop[6] primarily support the Java programming language, generating tests in JUnit format. In other programming ecosystems, tools like PYNGUIN[7][18], which is designed for Python test generation, and JSEFT[8][22], which utilizes function-coverage and abstraction algorithms for JavaScript, are also prominent. However, the industrial application of these automated unit test generation tools is somewhat restricted due to the substantial time investment required to analyze their outputs [10].

LLMs have assumed a significant role in test generation driven by Natural Language Processing (NLP) [37]. The advent of automatic code generation via LLMs offers substantial potential to reduce the time and costs associated with manual coding processes [13]. By training on human language inputs, LLMs such as OpenAI's Codex[4] can generate code snippets, documentation, and even repair bugs. This functionality has been extended in applications like GitHub Copilot, which has been extensively studied for its capability in test code generation [7, 27]. Despite these advancements, there remains a gap in the evaluation of the quality of code generated by these models.

3 STUDY DESIGN

3.1 Goals and Research Questions

This study aims to investigate potential irregularities and quality violations in test code generated by GitHub Copilot. Python was chosen for this investigation due to its extensive support by Codex [4]. GitHub Copilot was selected as the primary LLM owing to its broad integration with Integrated Development Environments (IDEs) and its capability to access and interpret code files [36]. To achieve these goals, three research questions were formulated:

- **RQ_1:** *How often are test smells detected in the Python test codes generated by GitHub Copilot?* This question aims to determine the incidence and frequency of test smells identified in the test codes generated by GitHub Copilot.
- **RQ_2:** *What types of test smells are detected in the Python test code generated by GitHub Copilot?* This question seeks to identify the most common types of test smells and to discern any recurring patterns in the test codes generated by Copilot.
- **RQ_3:** *How do practitioners feel about the quality of the test codes generated by GitHub Copilot?* This question aims to qualitatively assess practitioners' opinions on the quality of the test codes generated by Copilot, and to understand whether their perceptions align with the findings from the previous research questions.

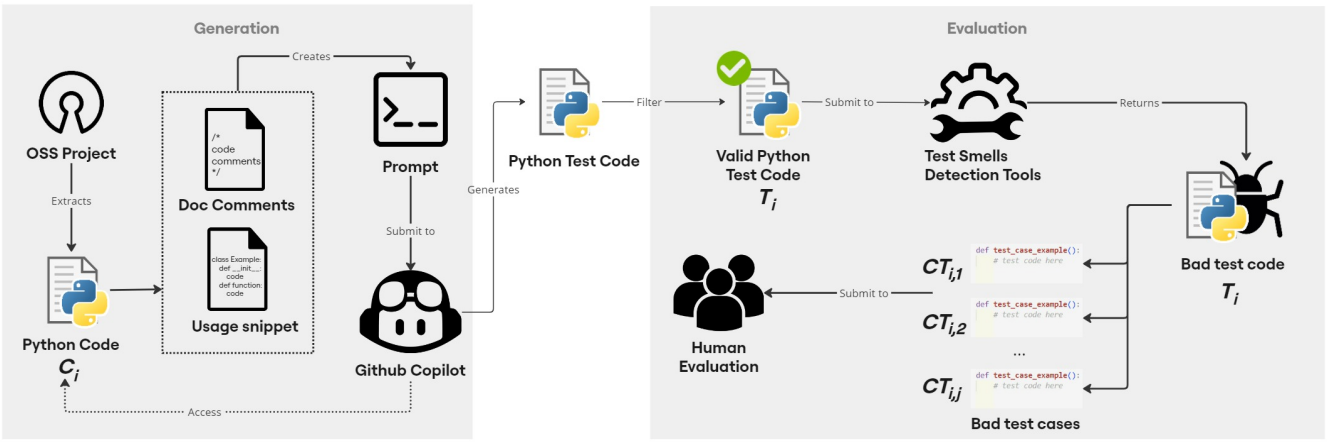
3.2 Study Settings

We split this study into four phases. First, we selected open-source projects. Next, we employed GitHub Copilot to generate test code for these projects. In the third phase, we utilized test smell detection tools to identify potential quality violations in the generated test code, addressing **RQ_1** and **RQ_2**. Finally, we conducted a survey among practitioners to gather their perceptions regarding the quality of the generated tests, thereby addressing **RQ_3**. The procedures are detailed below and illustrated in Figure 1.

3.2.1 Project Selection. For this study, we selected 4 open source Python projects from GitLab [9] and GitHub[10], coming from different domains, sizes and complexities. We considered the following criteria when selecting projects for this study. The chosen projects should be active, with recent commits, and, following El Haji et al. [7]'s recommendation, we decided less popular projects. This is because Codex has already been trained on source codes from the

[4]https://junit.org/junit5/
[5]https://www.evosuite.org/
[6]https://randoop.github.io/randoop/
[7]https://www.pynguin.eu/
[8]https://github.com/saltlab/JSeft
[9]https://about.gitlab.com/
[10]https://github.com/

Figure 1: Methodological procedures



most popular open-source projects and using small and medium-sized projects allows for more complete LLM training. Furthermore, we chose projects that already had unit tests to facilitate the identification of the most testable and critical parts of the production code. However, existing unit tests were disregarded for Codex training, and only production codes were used. This approach aims to ensure a more complete, accurate and targeted analysis of the test codes generated by GitHub Copilot without influence from those tests already present in the project. The training was carried out individually for each project, aiming to minimize information conflicts and the occurrence of hallucinations [20] during test generation. Table 1 contains all the projects used in this study and their respective information.

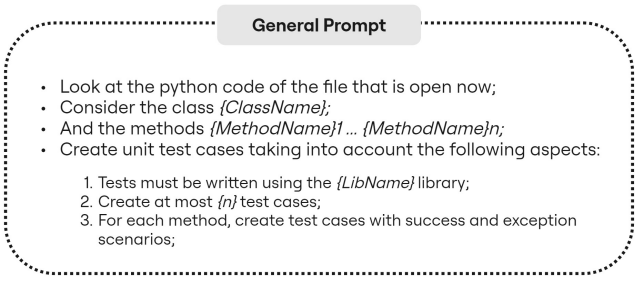
Table 1: List of open-source projects used in this study

Project	Domain	# LOC	# Classes	# Functions
python-lottie	File manipulation	20770	442	1573
click	CLI	10157	67	508
pyflunt	Domain Driven Design (DDD)	2969	14	200
brutils-python	Python Library	2849	25	140

3.2.2 Test Code Generation. Once the open-source projects had been selected, the production codes were chosen as the basis for generating the unit tests. To do this, Python files containing classes, methods or functions with testable returns were considered. Thus, for each production code file C_i, a test Python file T_i was generated. For each pair (C_i, T_i), GitHub Copilot generated the test case $CT_{i,j}$, with j being able to vary from 5 to 20 test cases per file. The libraries used to generate the test codes were pytest[11] and unittest[12].

To generate each test file T_i, we submitted a series of prompts so that Copilot could generate the tests according to the specifications. Each prompt should contain information about (i) which classes, methods, or functions would be tested, (ii) the types of scenarios to consider (successful, alternative, and exception), (iii) possible restrictions to be taken into account, such as the maximum number of test cases, the name of the test case and the parts of C_i to be disregarded, and (iv) the test library that would be used. Figure 2 shows a generic example of the prompt created:

Figure 2: General prompt used to generate the test files



After submitting the command prompt to Copilot, the unit tests were returned and stored in each project's directory for later execution. A sample of 30 pairs (C_i, T_i) was generated, totaling 397 test cases. All the test cases were executed, and only the valid ones were considered for this study. We considered valid code to be that which Python interpreted without any errors or hallucinations. After filtering, the sample was reduced to 194 valid test cases.

3.2.3 Test smells detection tools. Once the test cases had been generated and filtered, a series of Python-specific tools were selected to detect test smells in the code. Three different tools were used for this analysis: Pynose [33], TEMPY [8] and pytest-smell [3]. Each of them has its own techniques and approaches for identifying and categorizing test smells, providing a broader view of the quality of the test code generated by GitHub Copilot. We call *bad test codes* those Python T_i files in which at least one test smell has been identified in the corresponding test cases, which we call *bad test cases*. Table 2 lists the test smells that were addressed in this study and the tools that identified them. The list containing all the test cases and detections is available at the end of this paper.

[11] https://docs.pytest.org/en/8.2.x/
[12] https://docs.python.org/3/library/unittest.html

Table 2: Test smells detected by tools

Test smell	Tools	Description
Assertion Roulette	Pynose, pytest-smell	Several asserts without any explanation or message [33]
Magic Number Test	Pynose, pytest-smell	Existence of literal numeric values in a test [33]
Unknow Test	TEMPY	Tests without assertions [8]
Conditional Test Logic	Pynose, pytest-smell, TEMPY	Tests with control statements (if, for, while...)[33]
Eager Test	pytest-smell	Tests that invoke multiple methods of production code[3]
Duplicate Assert	pytest-smell	Duplicate assertions in the same test [3]
Test Maverick	Pynose	If the test suite has a fixture with setup, but a test case in this suite does not use this setup[33]

3.2.4 Practitioners' Assessment. Once the tools had detected the test smells, an evaluation was carried out by professionals on selected test cases. The test cases with the highest number of detected test smells were chosen, with the aim of investigating the professionals' perception of these tests generated by GitHub Copilot and checking whether their opinions corroborated with the results obtained by the tools. An online questionnaire was applied, containing six sets of test cases, called S_i, so that professionals could assess the quality of each set. Each set contained 2 to 3 test cases from the same T_i file. The professionals were given a list of characteristics corresponding to the description of each test smell detected by the tools (column *Description* of Table 2) and had to mark which of these characteristics they could identify in each set of test cases. They also had the opportunity to describe other characteristics not mentioned and suggest improvements to the code. To improve the validity of the evaluations, we do not inform practitioners that quality problems have been detected in the tests by the tools. A total of 20 professionals, mostly Software Developers or Software Quality Assurance with 1 to 7 years of experience in the industry, took part in the evaluation. Table 3 shows the profile of each professional who took part in this study.

Table 3: Practitioners' Profile

Practitioner	Experience	Load
P1	2 to 5 years	Quality Assurance (QA)
P2	1 to 2 years	Project manager
P3	Less than 1 year	Software Developer
P4	1 to 2 years	Software Developer
P5	2 to 5 years	Software Developer
P6	Less than 1 year	Quality Assurance (QA)
P7	2 to 5 years	Quality Assurance (QA)
P8	More than 7 years	Software Developer
P9	More than 7 years	Software Developer
P10	2 to 5 years	Quality Assurance (QA)
P11	1 to 2 years	Quality Assurance (QA)
P12	2 to 5 years	Quality Assurance (QA)
P13	1 to 2 years	Quality Assurance (QA)
P14	1 to 2 years	Software Developer
P15	2 to 5 years	Software Developer
P16	2 to 5 years	Software Developer
P17	1 to 2 years	Software Developer
P18	2 to 5 years	Software Developer
P19	2 to 5 years	Software Developer
P20	More than 7 years	Software Developer

4 RESULTS AND DISCUSSION

4.1 RQ$_1$: Test smell detection frequency

To determine the frequency, we recorded how many times a tool identified the presence of a test smell in a test case. We counted the detections of all three tools. In some cases, more than one tool identified the same test smell in the same test case. To avoid double counting, in this scenario we only considered one detection.

Of the 194 test cases analyzed, 92 had at least one occurrence of test smell, representing 47.4% of the total. Of the 30 test files generated, 21 contained at least one test smell detection. We also measured the frequency according to the number of detections in the same test case and in the same Python file. From this, we observed that the presence of a single test smell per test case is more common, while the occurrence of up to 3 test smells in the same test case is less frequent. Table 4 shows the three statuses detected in the test set generated by Copilot, the detection frequency for each test case $CT_{i,j}$ and for each test file T_i.

Table 4: Test smells detection frequency

Detection Status	$CT_{i,j}$	$\%CT_{i,j}$	T_i	$\%T_i$
1 test smell detected	84	43,2%	15	60%
2 test smell detected	6	3,1%	4	16%
3 test smell detected	2	1,1%	2	8%
TOTAL	**92**	**47,4%**	**21**	**84%**

Answer to RQ$_1$: 47.4% of the test cases generated by Copilot had at least one test smell detected by the tools. This means that in most of Python test files (84%), at least one test smell was identified in the corresponding test cases. These results show that code quality violations were identified recurrently in the tests generated by Copilot and that, in some tests, more than one violation can be found. This shows an instability in the quality of the test codes.

4.2 RQ$_2$: Types of test smells detected

To answer this question, we counted the occurrences of test smells according to their type. Table 5 shows the detection count for each type of test smell. The *Occurrences* column shows the count of occurrences of the type of test smells. *% Occurrences* highlights the percentage of these smells. *% Frequency* shows the frequency of detection of each type of test smell in relation to the 194 test cases generated.

Table 5: Occurrences of test smells by type

Test smell	Occurrences	% Occurrences	% Frequency
Assertion Roulette	82	75%	42%
Magic Number Test	11	10%	5,6%
Conditional Test Logic	5	4,5%	2,5%
Unknow Test	4	3%	2%
Duplicate Assert	3	2,7%	1,5%
Eager Test	3	2,7%	1,5%
Test Maverick	1	0,9%	0,5%

Table 5 reveals that the *Assertion Roulette* test smell was the most prevalent, occurring in 82 of the 92 test cases where test smells were detected. Next, the *Magic Number Test* smell was identified in 11 cases, followed by *Conditional Test Logic* and *Unknown Test*, with 5 and 4 occurrences respectively. These results, in light of

the concepts proposed by van Deursen et al. [32], indicate violations in the tests generated by Copilot, including the excessive use of undocumented assertions *(Assertion Roulette)*, the presence of numeric literal values *(Magic Number Test)* and the complexity of conditional logic *(Conditional Test Logic)*. In addition, the tools also identified test cases without assertions *(Unknown Test)*, duplicate test cases *(Duplicate Assert)* and test cases that make excessive calls to methods in production code *(Eager Test)*.

Answer to RQ$_2$: The tests generated by GitHub Copilot revealed a diversity of detected test smells. The most common test smell was *Assertion Roulette*, observed in 42% of the tests, suggesting an excessive use of undocumented assertions. In addition, other test smells were also identified, such as *Magic Number Test, Conditional Test Logic, Unknown Test, Duplicate Assert, Eager Test and Test Maverick*, although less frequently. These results highlight specific areas in which the test codes generated by Copilot can be improved.

4.3 RQ$_3$: Practitioners' Perceptions

To answer **RQ$_3$**, we performed a qualitative and quantitative analysis of the evaluations provided by industry professionals in the online questionnaire. In the quantitative analysis, we counted how many times each test smell was identified by professionals in each set of test cases. Table 6 presents the types of test smells included in the questionnaire. The columns referring to S_i denote the six sets of test cases evaluated and the absolute frequency with which professionals in their evaluations identified each test smell. Figure 3 presents a visual sample of the percentage of identification of each test smell in the evaluation.

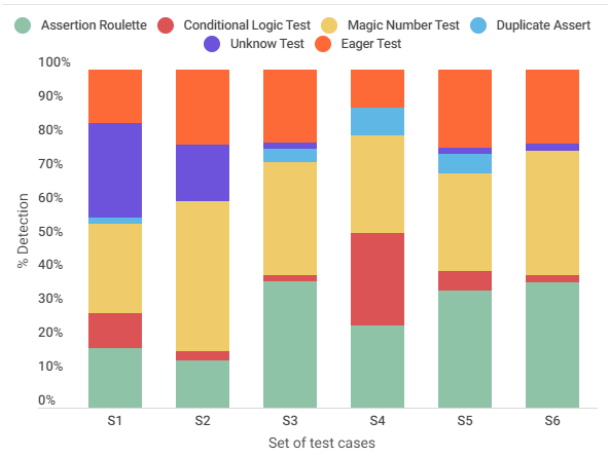
Table 6: Test Smells detected by practitioners

Test Smell	S_1	S_2	S_3	S_4	S_5	S_6
Assertion Roulette	10	5	19	15	18	17
Conditional Logic Test	6	1	1	17	3	1
Magic Number Test	15	16	17	18	15	18
Duplicate Assert	1	0	2	5	3	0
Unknow Test	16	6	1	0	1	1
Eager Test	9	8	11	7	12	10

From the analysis of Table 6 and the graph in Figure 3, we can see that the *Assertion Roulette* and *Magic Number Test* smells identified by the tools continued to be frequently detected by professionals. *Magic Number Test* was identified in 20 to 40% of the cases evaluated, while the *Assertion Roulette* appeared in 10 to 30% of the detections. *Eager Test* was also frequently cited, ranging from 15 to 20%. We noticed that some test smells prevailed in certain sets of tests. For example, *Unknown Test* was identified with considerable frequency only in the S_1 and S_2 sets, while *Conditional Test Logic* was detected more frequently only in the S_4 set. These observations indicate that certain smells are more likely to appear in specific contexts, reflecting the varied nature of the quality problems identified by professionals in the tests generated by Copilot.

Qualitatively, the professionals also offered opinions and suggestions on the quality of the test codes that went beyond the test smells included in the questionnaire. We noticed from the responses that some professionals identified very similar tests due to the naming of the variables in the methods being the same or showing only minimal differences. This similarity, along with other problems,

Figure 3: % Test smells detected by professionals for each set of test cases



made it difficult for some to distinguish between two or more tests. Several other problems related to readability were also identified, accompanied by suggestions for improvement. Some reports on these problems included lack of variables, repetition of code structure in more than one test, lack of context in the naming of tests, objects and variables, among others. In addition, professionals pointed out that the tests generated by Copilot did not take advantage of most of the resources offered by the *pytest* and *unittest* libraries, which may have contributed to the problem of code readability. Some reports on these problems were left on the form:

P9:"It is interesting to assign the positional parameters of the classes to variables, in order to facilitate the understanding of the meaning of each argument passed. Just by reading the test, you can't debug its purpose"

P11:"Pytest has some features that could make this test more efficient, and the same goes for the previous one."

P12:"You can use asserts such as assertTrue, assertEquals, etc. These would make the conditions simpler."

Answer to RQ$_3$: The practitioners' perception of the quality of the Python test codes generated by GitHub Copilot is predominantly negative in the six sets of test cases that were evaluated, highlighting several areas of concern. From the quantitative analysis, the professionals frequently identified test smells such as *Assertion Roulette, Magic Number Test*, and *Eager Test*, as well as others such as *Unknown Test* and Conditional Test Logic in specific contexts. Qualitatively, the professionals also pointed out significant problems related to the readability, maintainability and clarity of the tests generated. In addition, they pointed out that the tests generated did not use most of the resources, such as assertions and functions, offered by the *pytest* and *unittest* libraries. These observations indicate that although Copilot is capable of generating functional tests, there are significant areas for improvement, especially in terms of the clarity and readability of the test codes.

5 THREATS TO VALIDITY

Internal Threats: Although projects with recent commits have been chosen, this does not guarantee that these commits are substantial or relevant to test generation. The methodology of disregarding existing unit tests may have affected the way Copilot generated new tests, as it had no basis for how to build them, possibly introducing a form of bias. To mitigate this threat, we sought to identify exactly the pieces of code that the existing tests covered when generating the new tests.

External Threats: The analysis of 197 test cases in 30 files may not be sufficient to generalize the results to all tests generated by Copilot in different contexts and domains. The professionals who took part in the study may not represent the full diversity of the software industry, and there may be geographical and experience restrictions. The quality of the test codes generated by GitHub Copilot may depend on the specific version of the tool used, as well as possible future updates that may impact the results.

Construction Threats: The tools used (Pynose, TEMPY, pytest-smell) may have different approaches and algorithms for detecting test smells. The metrics used to assess the quality of tests, such as the count of test smells, may not capture all the general aspects surrounding the quality of test code. That's why we also decided to collect qualitative information from professionals. However, the perceptions of professionals are subjective and can vary widely depending on the experience and context of each professional.

6 RELATED WORK

El Haji et al. [7] conducted an experiment with the Codex [4] version of GitHub Copilot, investigating the usability of different test cases generated using command prompts. They found that a comment combining instructive natural language with an example of code usage resulted in more usable test generations. Similarly, Yu et al. [37] analyzed ChatGPT's ability to generate mobile test scripts. This study's findings indicate that ChatGPT can generate useful tests when provided with sufficient context and information about the project architecture. Although several studies [27, 29] have found promising results on the capability and limitations of LLMs, they did not evaluate the design quality of the generated test code, focusing mainly on the validity of these codes. For this study, we used the test generation methods of the two works [7, 27].

Several studies have already been conducted by researchers to assess its quality for production code. Yetistiren et al. [35] examined the quality of the code produced by GitHub Copilot, looking at aspects such as efficiency and design. They observed that although Copilot is able to generate valid code, many of them still have problems related to efficiency and design. In complementary research, Yetiştiren et al. [36] investigated the presence of code smells in the codes generated by Amazon's CodeWhisperer, Copilot and ChatGPT. They found that certain code smells tend to recur in the generated code and that the LLMs themselves have the ability to fix these maintenance problems. This pattern of repeating problems that impact code maintainability was also identified by Hansson and Ellréus [13], but the authors emphasized that ChatGPT and GitHub Copilot have proven to be more effective in generating quality code. Thus, although there have been several evaluations of the quality of code produced by LLMs, these analyses have focused mainly on production code and have not specifically addressed test code. From this works we were able to extract code quality analysis techniques and adapt them to test code.

7 CONCLUSION AND FUTURE WORK

In this paper, we conducted an empirical study on the quality of test code generated by GitHub Copilot for Python. To do this, we designed three research questions and generated a total of 194 valid test cases in 30 Python files to analyze the quality of the code through the detection of test smells carried out by both automated tools and development and quality assurance professionals.

We analyzed the frequency with which the tools detected smells in the sample of test cases generated and found that approximately 47% of the test cases contained at least one test smell, which indicated the presence of smells in 84% of the Python files that contained these test cases. In addition, we investigated which types of test smells were most commonly found by the tools, finding that *Assertion Roulette* was the most prevalent. Finally, we analyzed the perception of professionals about these test cases, revealing aspects other than those reported by the tools, such as readability problems, repetition of code and failure to take advantage of the resources offered by the libraries. As future work, we intend to further explore the test generation techniques in GitHub Copilot to investigate whether the way they are generated has an impact on the quality of the code that the LLM generates. We also want to analyze these aspects in other programming languages and with others LLMs.

ARTIFACT AVAILABILITY

We provide our artifacts at: https://zenodo.org/records/11426592

ACKNOWLEDGEMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; CNPq grants 315840/2023-4 and 403361/2023-0; and FAPESB grantPIE0002/2022.

REFERENCES

[1] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27

[2] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. In *When, how, and why developers (do not) test in their IDEs* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 179–190. https://doi.org/10.1145/2786805.2786843

[3] Alexandru Bodea. 2022. Pytest-Smell: a smell detection tool for Python unit tests. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Virtual</city>, <country>South Korea</country>, </conf-loc>) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 793–796. https://doi.org/10.1145/3533767.3543290

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario

Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

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

[6] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 201–211. https://doi.org/10.1109/ISSRE.2014.11

[7] Khalid El Haji, Carolin Brandt, and Andy Zaidman. 2024. In *Using GitHub Copilot for Test Generation in Python: An Empirical Study* (Lisbon, Portugal) *(AST '24)*. ACM, New York, NY, USA, 11. https://doi.org/10.1145/3644032.3644443

[8] Daniel Fernandes, Ivan Machado, and Rita Maciel. 2022. TEMPY: Test Smell Detector for Python. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering* (<conf-loc>, <city>Virtual Event</city>, <country>Brazil</country>, </conf-loc>) *(SBES '22)*. Association for Computing Machinery, New York, NY, USA, 214–219. https://doi.org/10.1145/3555228.3555280

[9] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[10] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 24, 4, Article 23 (sep 2015), 49 pages. https://doi.org/10.1145/2699688

[11] Danielle Gonzalez, Joanna C.S. Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. 2017. A Large-Scale Study on the Usage of Testing Patterns That Address Maintainability Attributes: Patterns for Ease of Modification, Diagnoses, and Comprehension. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 391–401. https://doi.org/10.1109/MSR.2017.8

[12] D. Graham, R. Black, and E. van Veenendaal. 2021. *Foundations of Software Testing ISTQB Certification, 4th edition*. Cengage Learning. https://books.google.com.br/books?id=mOwxEAAAQBAJ

[13] Emilia Hansson and Oliwer Ellréus. 2023. Code Correctness and Quality in the Era of AI Code Generation : Examining ChatGPT and GitHub Copilot. , 69 pages. https://lnu.diva-portal.org/smash/record.jsf?pid=diva2%3A1764568&dswid=9049

[14] V. Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns: Effective testing styles, patterns, and reliable automation for unit testing, mocking, and integration testing with examples in C#*. Manning. https://books.google.com.br/books?id=CbvZyAEACAAJ

[15] Dong Jae Kim. 2020. An Empirical Study on the Evolution of Test Smell. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 149–151.

[16] Dong Jae Kim, Tse-Hsun Chen, and Jinqiu Yang. 2021. The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering* 26 (2021).

[17] Chun Li. 2022. In *Mobile GUI test script generation from natural language descriptions using pre-trained model* (Pittsburgh, Pennsylvania) *(MOBILESoft '22)*. Association for Computing Machinery, New York, NY, USA, 112–113. https://doi.org/10.1145/3524613.3527809

[18] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: automated unit test generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 168–172. https://doi.org/10.1145/3510454.3516829

[19] P. Maragathavalli. 2011. Search-based software test data generation using evolutionary computation. *ArXiv* abs/1103.0125 (2011). https://api.semanticscholar.org/CorpusID:25209645

[20] Ariana Martino, Michael Iannelli, and Coleen Truong. 2023. Knowledge Injection to Counter Large Language Model (LLM) Hallucination. In *The Semantic Web: ESWC 2023 Satellite Events*, Catia Pesquita, Hala Skaf-Molli, Vasilis Efthymiou, Sabrina Kirrane, Axel Ngonga, Diego Collarana, Renato Cerqueira, Mehwish Alam, Cassia Trojahn, and Sven Hertling (Eds.). Springer Nature Switzerland, Cham, 182–185.

[21] Phil McMinn. 2004. Search-based software test data generation: a survey: Research Articles. *Softw. Test. Verif. Reliab.* 14, 2 (jun 2004), 105–156.

[22] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. JSEFT: Automated Javascript Unit Test Generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102595

[23] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic Test Smell Detection Using Information Retrieval Techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 311–322. https://doi.org/10.1109/ICSME.2018.00040

[24] Zedong Peng, Xuanyi Lin, Michelle Simon, and Nan Niu. 2021. Unit and regression tests of scientific software: A study on SWMM. *Journal of Computational Science* 53 (2021), 101347. https://doi.org/10.1016/j.jocs.2021.101347

[25] P. Runeson. 2006. A survey of unit testing practices. *IEEE Software* 23, 4 (2006), 22–29. https://doi.org/10.1109/MS.2006.91

[26] Railana Santana, Luana Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. 374–379.

[27] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[28] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. 2019. On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 121–125. https://doi.org/10.1109/MSR.2019.00028

[29] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. arXiv:2305.00418 [cs.SE]

[30] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs.SE]

[31] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 4–15. https://doi.org/10.1145/2970276.2970340

[32] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. Refactoring Test Code. *Proceedings 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)* (may 2001).

[33] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2022. In *PyNose: a test smell detector for python* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 593–605. https://doi.org/10.1109/ASE51524.2021.9678615

[34] Tao Xie and David Notkin. 2006. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal* 13, 3 (July 2006), 345–371.

[35] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. In *Assessing the quality of GitHub copilot's code generation* (Singapore, Singapore) *(PROMISE 2022)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/3558489.3559072

[36] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. In *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. https://doi.org/10.48550/arXiv.2304.10778

[37] Shengcheng Yu, Chunrong Fang, Yuchen Ling, Chentian Wu, and Zhenyu Chen. 2023. In *LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities*. 206–217. https://doi.org/10.1109/QRS60937.2023.00029