Unit Test Generation with GitHub Copilot

A Case Study

Faculty of Technology

Information and Communication Technology, Software Engineering

Master of Science in Technology thesis

Author:

Sami Humalajoki

Supervisor:

Antero Järvi

16.6.2024

Turku

Master of Science in Technology thesis

Subject: Software Engineering
Author: Sami Humalajoki
Title: Unit test generation with GitHub Copilot, A Case Study
Number of pages: 63 pages
Date: 16.6.2024

Artificial intelligence has taken remarkable steps in recent years. Natural language processing technology and large language models have changed many aspects of software development process. Different tools have been developed to aid in the software development process. In this study we aim to evaluate GitHub Copilot's abilities in automated unit test generation. The study is motivated by the critical role of testing in software development. Effective testing ensures code quality, reliability, and maintainability. Software systems have grown increasingly complex and with it the demand for efficient test generation tools. This study aims to assess GitHub Copilot's abilities in real-world software development standards of testing by introducing unit tests to a legacy software system.

Four research questions guide the evaluation: the immediate usability of Copilot's test suggestions based on their compilation success rates, the correctness of these suggestions through execution error analysis, the effectiveness of test suggestions measured by code coverage, and the presence of test smells indicating potential maintainability problems. The research methodology employs an iterative one-shot method to evaluate GitHub Copilot's performance in generating and refining test cases, structured into three primary steps. First, test cases are generated by prompting Copilot within an IDE, followed by verifying and correcting their syntactic accuracy using IDE feedback and Copilot's correction suggestions. Finally, the syntactically correct test cases are executed, corrected as needed, and assessed for functional correctness and quality metrics like code coverage and test smells.

The research findings indicate that GitHub Copilot can generate valid unit tests, but its performance is inconsistent and frequently requires human intervention. Copilot struggles with complex mocking scenarios, often fails to detect straightforward errors, and relies heavily on the provided context, leading to potential reliability issues. Code coverage analysis shows that Copilot is effective in straightforward testing scenarios, achieving high coverage in simple methods, but performs poorly with methods of high cyclomatic complexity. Additionally, Copilot's tests exhibit common test smells such as Magic Number Tests and Lazy Tests, which are more common in complex code, suggesting a preference for speed over quality and a tendency to overlook best practices in unit testing. Overall, while Copilot can produce reasonable quality tests, its effectiveness diminishes with increased code complexity.

The results indicate the need of frequent of human intervention for error correction and test quality enhancement. Also, the presence of common test smells may indicate a preference for speed over best practices. Copilot might also benefit from internal feedback system, where it could execute and assess its code suggestions. These insights suggest that Copilot is valuable for straightforward testing scenarios, but its reliability decreases with more complex code.

Key words: artificial intelligence, natural language processing, large language models, software testing, unit testing.

Table of contents

1 Introduction 1

2 Unit Testing 3

2.1 Test-driven Development 3

2.2 Unit Testing 4

2.3 Unit Test Evaluation Metrics 5

2.3.1 Code Coverage 5

2.3.2 Test Smells 8

2.3.3 Flawed Test Cases 12

2.4 Best Practices in Unit Testing 13

3 Natural Language Processing and LLM-based Software Engineering 17

3.1 Natural Language Processing Techniques in Software Engineering 17

3.2 Overview of Large Language Models 18

3.3 LLMs Capabilities and Limitations in Software Engineering 21

3.3.1 Capabilities 21

3.3.2 Limitations 22

4 Unit Test Generation with Large Language Models 24

4.1 Research Review 24

4.2 Related Work 25

4.2.1 Coverage 27

4.2.2 Correctness 28

4.2.3 Test Smells 30

4.2.4 Research on Test Generation with GitHub Copilot 32

5 Research Settings and Methods 35

5.1 System Description 35

5.2 Research Questions 37

5.3 Methodology 38

5.4 Results 40

5.4.1 Validity and Correctness 40

5.4.2 Code Coverage 46

5.4.3 Test Smells 49

6 Conclusions **53**

6.1 Summary of Findings **53**

6.1.1 Validity and Correctness 53

6.1.2 Code Coverage 54

6.1.3 Test Smells 55

6.2 Future Research **56**

6.3 Implications **56**

References **58**

1 Introduction

Recent advancements in artificial intelligence (AI) and natural language processing (NLP) are changing the software development industry in many different aspects. With AI-powered code completion tools, such as GitHub Copilot, code generation and testing can be automated to a degree. Copilot suggests code snippets, entire functions, and even test cases, which helps to streamline the software development process and possibly increase productivity. But the usability and reliability of Copilot's test generation features still need examination in real-world settings.

The motivation for this study comes from the importance of testing in software development. In the case study we are adding unit tests to a legacy system in order to apply better with the test-driven development (TDD) approach. Effective testing ensures code quality, reliability, and maintainability. Thorough testing prevents bugs and reduces long-term maintenance costs. As software systems become more complex, the need for efficient and accurate test generation tools grows. GitHub Copilot can provide a good opportunity to improve testing approaches. It is important to assess whether Copilot can meet the standards required for practical use in real-world software development. Evaluating Copilot's ability to generate valid, correct, and comprehensive test cases provides understanding of its strengths and limitations, which in turn gives insights into the ways the tool can be integrated into the development workflow.

This study focuses on identifying common errors, execution issues, code coverage, and test smells in Copilot's suggestions, addressing critical aspects of test quality and maintainability. We explore how GitHub Copilot generates and refines test cases through four key research questions. First, we assess Copilot’s test suggestions by examining their compilation success rates, identifying common errors like syntax errors and type mismatches. Through this their immediate usability can be assessed. The second question evaluates the correctness of Copilot’s test suggestions by analyzing execution errors, such as exceptions or incorrect logic. We run the tests, record errors, and assess if they validate intended functionality. The aim is to determine their reliability. Next, we measure code coverage to evaluate the effectiveness of Copilot’s suggestions in testing various parts of the codebase. Finally, we identify and analyze test smells in Copilot’s suggestions, indicators of potential problems like excessive setup code or overly complex logic. This analysis provides insights into the quality and maintainability of Copilot’s suggestions, offering areas for improvement.

To address these research questions, the methodology follows a structured procedure in three main steps. Initially, Copilot generates test cases, which are then saved for evaluation. Next, the correctness of these tests is assessed, and only viable ones proceed to execution. Finally, successful tests are evaluated for quality, while failed ones are corrected and re-evaluated. This process is repeated to explore Copilot's error correction ability, with the goal of determining overall test quality and maintainability.

The structure of this thesis is designed to provide a comprehensive theoretical understanding of unit testing, NLP, and large language models (LLMs) before moving to describing the study in detail. Chapter 1 introduces the thesis by explaining its aims, scope, and importance. Chapter 2 explores unit testing, starting with TDD principles and techniques. It discusses the purpose and techniques of unit testing, evaluation metrics like code coverage, and common issues in test cases. Recommended practices for creating effective unit tests are also provided. Next in Chapter 3 we discuss the use of NLP and LLMs in software engineering, covering their applications, strengths, and weaknesses. It includes an overview of LLM architecture and capabilities, as well as challenges associated with their use. We then move to reviewing existing research on LLMs in software engineering, focusing on unit test generation in Chapter 4. It explores their impact on code coverage, test accuracy, and the occurrence of test smells. A case study on GitHub Copilot's use for test generation is also presented. Chapter 5 presents the original research conducted for the thesis, describing the research system, tasks, and methodology. It presents findings on LLM-generated test accuracy, coverage, and test quality. Lastly in Chapter 6 we summarize the key findings, discuss their implications, and suggest future research directions.

In closing, this thesis looks into how software development is evolving, especially with new AI and NLP technologies like GitHub Copilot. While Copilot offers ways to automate code writing and testing, its reliability in real-world situations needs careful consideration.

2 Unit Testing

In this chapter the theoretical motivation of this thesis is discussed. First, TDD is introduced. The approach is the primary motivation behind the research that is conducted in this thesis. Next unit testing is discussed. Unit testing is one of the main methods of TDD. Metrics used to evaluate the efficacy of unit testing are also discussed. Lastly, best practices in unit test generation are summarized.

2.1 Test-driven Development

In this study we are introducing unit testing to legacy code. The motivation behind this is to apply more of the principles of TDD to the software in question. TDD is a practice where code is written after the test (Beck, 2022). The aim of the practice is to produce better quality code that has fewer defects. According to Madeyski (2010) TDD provides instant feedback on whether a new functionality has been implemented as intended and whether it interferes with previously implemented functionality. It encourages developers to break down problems into small, manageable programming tasks to enhance productivity. TDD enforces keeping tests up to date, which enable avoiding complexities in the code through continuous refactoring. Running tests frequently helps ensure a certain level of quality and test coverage. Additionally, tests provide context for making low-level design decisions, such as naming classes and methods and defining interfaces. They also serve as a form of communication and documentation, showcasing concrete examples of how to exercise a class's functionality which also provides knowledge to new participating developers, hence encouraging refactoring and maintenance activities (Madeyski, 2010).

According to Madeyski (2010) the TDD approach also offers a new perspective on software product quality by considering the quality of test code. The quality of tests can indicate the quality of the related production code, especially when writing tests is an integral part of the development practice. Turhan et al. (2010) found that adopting TDD in the software development process may improve internal and external quality of code, but this is dependent on the evaluation metric. Complexity is often reduced, but better cohesion is oftentimes not achieved. The TDD approach is said to increase productivity (Turhan et al., 2010), but as Madeyski (2010) points out the process of adopting TDD is labor intensive. It may even decrease productivity in the initial stages of adoption. However, Madeyski (2010) concludes that despite the initial drop in productivity, continuous testing and refactoring can eventually increase development speed. According to Acharya (2014) in TDD code is only written to satisfy a test and the code is refactored to improve its quality. The test-first practice is also thought to increase the maintainability and reliability of the code (Tosun et al., 2018).

2.2 Unit Testing

Unit testing is one of the testing practices of the TDD approach. The classical definition of a unit test is that it is a piece of code, typically a method, that calls another piece of code and verifies the accuracy of certain assumptions (Saleh, 2013). Acharya (2014) describes unit testing as performing sanity check of code, i.e.. checking whether the software produces coherent responses. A good unit test should possess several key characteristics.

Firstly, it should be automated, meaning that it can be executed automatically without manual intervention. This allows other developers to easily repeat the test for every significant code change. Secondly, a unit test should be repeatable. This means that it consistently produces the same results when executed multiple times. By ensuring repeatability, any changes or issues in the code can be identified and addressed promptly.

Furthermore, a unit test should be easy to understand. It should be clear and comprehensible to other developers, enabling them to grasp the purpose and functionality of the test. This facilitates collaboration and encourages the addition of new test cases or updates to existing ones. In addition, a good unit test should be incremental. This implies that the test should be updated whenever a new relevant defect is detected in the code. By continuously improving the test, the likelihood of recurring defects is minimized. Also, a unit test should be easy to run. It should be executable with a simple command or by clicking a button. The execution time of the test should also be relatively short, as fast unit tests contribute to the overall productivity of the development team. (Saleh, 2013)

Saleh (2013) argues that unit testing is not just a nice-to-have, but a mandatory activity in software engineering. It plays a crucial role in ensuring the success and stability of software solutions, especially when dealing with changes over time. One of the key advantages of unit testing is that it simplifies the integration of different components within a system. Without proper unit testing, the process of tracing defects and identifying problematic components becomes complex and time-consuming. This can lead to ineffective use of resources. (Saleh, 2013)

Furthermore Saleh (2013) argues that unit testing helps manage the number of new defects and regression defects that arise as the code base becomes more complicated. By having repeatable test cases, developers can ensure that resolved defects do not reappear after subsequent code changes. This significantly improves the quality of the software and reduces the time spent on testing during each deployment or phase. Unit testing also serves as a valuable reference for system documentation. It includes test scenarios for system use cases and demonstrates how system application programming interfaces (APIs) are utilized, reflecting the current design of the system. This makes unit testing an essential foundation for code and design refactoring, enabling further enhancements in the system. (Saleh, 2013)

2.3 Unit Test Evaluation Metrics

In this chapter metrics used to evaluate the efficacy of unit tests are discussed. One of the most common metrics used is code coverage, which has different aspects of evaluating coverage. The research in this thesis is evaluated by statement and branch coverage, which are discussed. One commonly used coverage metric is input space coverage. It refers to designing test cases that cover all possible inputs, including valid and invalid inputs as well as boundary values and combinations of different inputs. However, this metric is not used in the existing research literature to evaluate LLM's test generation performance, hence it is not considered in this research either. We are mainly interested in the correctness of the generated tests and how well the LLM is able to correct failing tests based on error message feedback. Another evaluation method is test smells. Some of the benefits and limitations of test smells are discussed and some tools that assess test smells are introduced. In this research tsDetect tool is used to assess the generated tests. The tool is discussed in further detail in Chapter 5.3. Lastly, best practices in unit testing are discussed to provide an outlook on the expected behavior of the LLM.

2.3.1 Code Coverage

Code coverage refers simply to the percentage of the code that the testing covers. There are different aspects that can be measured. In this work the two available code coverage values are statement and branch coverage. To measure these metrics, we use JaCoCo, an open-source Java code coverage library[1]. This tool is discussed in further detail in Chapter 5.3.

[1] https://github.com/jacoco/jacoco, visited 10.6.2024

Statement coverage is a basic part of software testing with several advantages. It provides a measurable way to check how complete the testing is. By analyzing statement coverage, testers can see how much of the source code the test suite has covered, which helps identify areas that need more testing. Statement coverage also helps find code that hasn't been run, highlighting parts that weren't covered during testing. This is important for finding dead code or areas that couldn't be reached because of logical mistakes or insufficient test coverage. (Cornett, 1996)

High statement coverage is also important for quality assurance. When it's achieved, it means a lot of the code has been tested, which can make us more confident in the software's quality. While it doesn't mean there are no defects, it does show that the testing has been thorough. The data from statement coverage can also be useful for debugging. If a test fails or something unexpected happens, it can help identify the parts of the code that were run before the failure, making debugging easier and helping to focus on areas that might have problems. (Hemmati, 2015)

Even though its useful, statement coverage has a few limitations that need to be thought about when evaluating unit testing. (Cai & Lyu, 2005). A significant limitation is that it only measures if individual lines of code have been run but doesn't give any information about how good or effective the tests are. This means that even if statement coverage is high, it can't ensure that all possible situations, edge cases, or combinations of inputs have been covered. (Jay et al., 2015)

Also, statement coverage might not work as well with complex logic paths in code, especially in loops and conditional statements. It's meant to make sure all the code is run, but it might miss some paths, leaving important parts of the code untested and open to unnoticed defects. Plus, focusing too much on statement coverage can lead testers to care more about getting high coverage percentages than about the quality and relevance of the tests. This "check-the-box" testing can hurt the testing process by not thoroughly checking the software's correctness and robustness. (Hemmati, 2015)

Statement coverage can help find code that hasn't been run, but it's not always good at finding all types of faults. Some mistakes, like semantic or integration errors, might not be found even when coverage is high, so different testing methods are needed to find all defects (Cai & Lyu, 2005). Trying to get 100% statement coverage is often not practical, especially with big and complex codebases. Things like external dependencies, behavior specific to a platform, and exceptional error conditions can make it hard to get complete coverage, making it not cost-effective to try to cover everything (Acharya, 2014).

Just using statement coverage to measure how complete the testing is can give a false sense of security. Even if coverage is high, there might still be defects or vulnerabilities that haven’t been found. So, it’s important to use other testing methods along with statement coverage to make sure the testing is thorough, and quality is assured. A variety of testing methods and measures make sure the testing is rigorous and quality is thoroughly checked.

Branch coverage in software testing helps give a comprehensive evaluation of how complete the testing is. It’s different from statement coverage because it gives a more detailed check, making sure not just every line of code is run, but also that different decision paths in the code are covered (Cornett, 1996). This complete approach helps find potential defects or unusual things related to conditional logic, which makes the testing more effective. Branch coverage is good at finding specific decision points in the code where not all possible outcomes have been looked at. This is key for finding what’s missing in the test suite and deciding where more testing is needed to fully cover these areas (Hemmati, 2015).

Branch coverage is a useful measure in software testing, but it has some limitations. One limitation is how it deals with decision coverage. Branch coverage checks if each branch in a decision statement has been run, but it doesn’t make sure all possible conditions in each decision have been tested. For example, getting branch coverage on an if-else statement doesn’t necessarily mean both the true and false conditions have been thoroughly tested, which shows there might be a gap in how complete the coverage is. (Wei et al., 2012)

Branch coverage faces challenges when dealing with complex logic scenarios. Code structures containing nested or intricate conditional statements can create multiple decision paths that are difficult to cover entirely using branch coverage alone. Achieving high branch coverage may require an impractical number of test cases or may be unattainable altogether. Additionally, achieving 100% branch coverage is often unrealistic, especially in complex codebases with intricate decision logic. Some branches may be hard to reach or require specific input conditions that are challenging to replicate in test cases, making comprehensive coverage difficult. (Wei et al., 2012) In summary, branch coverage is a useful metric for assessing testing thoroughness. However, it’s essential to combine it with other testing techniques and metrics to achieve comprehensive test coverage and ensure high-quality software.

Applying **both statement and branch coverage** metrics typically leads to better test coverage and improved quality assurance compared to using either metric in isolation. When untested code segments are identified and different types of scenarios and conditions are considered during testing, a more thorough investigation of the codebase makes the approach more comprehensive. A more detailed understanding of where to focus during testing can be achieved when statement and branch coverage are combined as these ensure that both every line of code and different decision paths are covered. When both statement and branch coverage are high, it demonstrates that different aspects of the code are tested, growing confidence in the reliability and quality of the system. Also, combining the two measures the risk of undiscovered defects and vulnerabilities can be reduced especially in critical parts of the code. Using both metrics also helps in debugging and troubleshooting as the root cause can be pinpointed more quickly and accurately. In conclusion, either metric alone can provide valuable insights, but combining them a more comprehensive and effective quality assurance is achieved. (Chekam et al., 2017)

2.3.2 Test Smells

Test smells are patterns or characteristics that can be found in the design, implementation, execution of software tests. They can indicate problems, weaknesses, or deficiencies in the software that is being tested. Test smells are similar to code smells (Tufano et al., 2016), which point to problems in source code. Test smells highlight areas of the code where effectiveness, maintainability, or reliability is compromised. They can manifest in different ways, such as overly complex test logic, redundant or duplicated tests, inadequate coverage, or they may be coupled to implementation details. Maintaining a robust and efficient testing strategy relies on identifying and addressing test smells. They may impact the test suite's reliability, maintainability, and effectiveness. (Deursen et al., 2001)

Test smells cover a range of issues that can affect how reliable and maintainable the test suite is. For example, *fragile tests* are prone to breaking when even small things change in the system or its environment, usually because they're too tightly connected to specific details of how things are implemented. Similarly, *brittle assertions* might lead to test failures over minor changes in system behavior, showing the need for stronger assertions. Problems like duplicated or conditional test logic, along with tests that are hard to understand, make it harder to maintain and work with the test suite. Using mocks or stubs too much can make test setups overly complicated and obscure what the system is really doing, making tests harder to keep up. *Eager test setup* and *lazy test verification* can also affect how well tests work and how much they cover. Things like *integration tests in disguise* or tests relying on *magic* values add extra complications and risks to the testing process. Trusting the results of tests that run slowly, are inconsistent, or are in any way unreliable is difficult. There may be issues with leaking test data, the tests try to do too much at once, or they make too many assumptions. This emphasizes the need for evaluation and fixing of such issues. Regularly reviewing and improving tests through refactoring and code review processes are essential for keeping the testing process strong and making sure the software being tested is of high quality and reliability, as noted by Kim et al. (2021). Different test smell types are described in Table 1 below.

Table 1. Different test smell types

Test Smell	Description
Fragile Test	Tests highly sensitive to changes in the system under test, often due to excessive coupling to implementation details.
Brittle Assertion	Tests with overly specific assertions about the system's behavior, prone to failure with minor changes in implementation.
Duplicate Test Logic	Duplication of test logic across multiple tests, leading to increased maintenance effort and decreased readability.
Conditional Test Logic	Tests with conditional logic (e.g., if statements) that obscure the test's intent and make it harder to understand and maintain.
Hard-to-Read Test	Tests with convoluted or unclear logic that is difficult to understand at a glance, impeding maintenance and comprehension.
Overuse of Mocks/Stubs	Excessive use of mocks or stubs in tests, resulting in overly complex test setups and potentially obfuscating the system's behavior.
Eager Test Setup	Tests performing excessive setup or configuration before exercising the system, leading to slower test execution and increased brittleness.
Lazy Test Verification	Tests lacking sufficient assertions or verification steps, resulting in incomplete validation of the system's behavior.
Integration Test in Disguise	Tests blurring the line between unit and integration tests by exercising multiple components or dependencies, leading to longer test execution times and increased fragility.
Resource Leak Test	Tests inadvertently leaking resources (e.g., memory, database connections) due to improper cleanup or teardown.
Magic Test Values	Tests relying on "magic" values (e.g., hard-coded constants) without clear explanation or justification, reducing maintainability.
Long-Running Test	Tests taking an excessive amount of time to execute, often due to complex setup or teardown procedures or inefficient test logic.
Flaky Test	Tests exhibiting non-deterministic behavior, producing different outcomes under identical conditions, often due to race conditions or timing issues.
Unstable Test	Tests failing intermittently without changes to the code or environment, making it difficult to trust their results.
Test Data Leakage	Tests inadvertently leaking sensitive or confidential data into test outputs or logs, posing security or privacy risks.
Non-Atomic Test	Tests relying on shared state or dependencies, leading to interference between tests and potential false positives or negatives.
Over-assertive Test	Tests with too many assertions, making it hard to isolate the cause of failures and increasing the likelihood of false positives.
Untestable Code	Code difficult or impossible to test due to excessive coupling, lack of modularity, or other design issues.

Benefits and limitations of test smells in assessing test quality

Using test smells as a tool to evaluate the effectiveness of unit tests offers several advantages. It helps detect issues early in the test suite, serving as a signal for potential weaknesses. This early detection allows developers to address problems before they become significant obstacles. It also helps enhance the overall quality of tests by highlighting areas needing attention. By focusing on improving individual tests, developers can make them clearer, easier to maintain, and more reliable. Additionally, test smells assist in managing the test suite by identifying duplication, complexity, or fragility. By addressing these issues, developers can make the test suite easier to understand, modify, and expand, reducing the effort needed to maintain tests and freeing up time for other tasks (Garousi & Kücük, 2018). Dealing with test smells increases confidence in test results, ensuring that tests are well-designed and thorough. It also guides targeted refactoring efforts, making improvements efficiently (Panichella et al., 2021). Following established best practices helps developers ensure that their tests meet industry standards. Overall, strategically using test smells helps developers identify and resolve issues, leading to a more reliable test suite and improved software quality (Garousi & Kücük, 2018).

There are also limitations in using test smells to assess the effectiveness of unit tests. Firstly, there's a subjective aspect to identifying test smells, which means different developers or teams may interpret them differently. This can lead to inconsistencies in evaluation (Tufano et al., 2016). False positives can also complicate the evaluation process. Not all instances of test smells indicate real issues in the test suite; some might be harmless or depend on the context, so sensitivity is needed to avoid unnecessary refactoring (Panichella et al., 2021). On the other hand, false negatives are significant shortcomings that might not show up as recognizable smells, potentially leading to important deficiencies in testing practices being overlooked (Panichella et al., 2021). Additionally, while test smells mainly focus on structural aspects of tests like duplication or complexity, they might miss issues related to test coverage, adequacy, or relevance, giving an incomplete assessment of the test suite (Panichella et al., 2021). Furthermore, the importance of a test smell can vary depending on the context of the software project, so it's important to understand its implications within specific domains or methodologies (Kim et al., 2021).

Test smell tools

Aljedaani et al. (2021) conducted a systematic review of different tools used for analyzing test smells. Their examination of these tools revealed several important findings. Firstly, while there's some overlap in the types of smells detected by various tools, there are differences in how they're implemented and defined. Most test smell detection tools are designed for Java systems using the JUnit framework. However, there's not much reporting on how accurate these tools are or how bias is handled in evaluating their quality. Creating a standard for validating test smell detection tools could improve trust and reliability of the tools. There is also a need to develop the tools to give appropriate refactoring suggestions and not act only as a way to detect test smells, as noted by Aljedaani et al. (2021).

In this study, we used tsDetect, an open-source software designed to find test smells in Java-based software systems. This is described in more detail in Chapter 5.3. It works by applying a set of preset rules to detect these smells in test code (tsDetect). Aljedaani et al. (2021) evaluated tsDetect's performance using a benchmark of 65 unit test files, which included examples of 19 different types of test smells. The results showed that tsDetect is highly accurate in detecting these smells, with an average precision score of 96% and an average recall score of 97%. In terms of correctness, tsDetect consistently identified test smells accurately, with precision scores ranging from 85% to 100% and recall scores ranging from 90% to 100%. The average F-score, which combines precision and recall, was found to be 96.5% (Aljedaani et al., 2001). These findings suggest that tsDetect is a highly effective tool for the detection of test smells within Java software systems. This level of effectiveness makes it a valuable asset in the current study to evaluate the quality of unit tests generated by LLMs.

2.3.3 Flawed Test Cases

Flawed test cases in unit testing come in various forms, highlighting potential weaknesses in the testing process. These cases can lead to inaccuracies and reduce the effectiveness of testing. One common type of flawed test case occurs when there are incorrect assumptions about how the code should behave. Such cases may produce inaccurate results, leading to errors in assessing software correctness. Flawed test cases often result from poorly defined or unclear test scenarios, making it difficult to establish accurate assessment criteria. These issues can stem from misunderstandings or ambiguities in requirements or expectations, complicating the testing process. (Mathur, 2008)

Misunderstanding the system requirements or specifications can lead to flawed test cases. If the requirements are not clear, they may be interpreted differently by stakeholders. This may lead to test cases that do not capture the intended software functionality. Making flawed assumptions about user behavior, such as inputs or interactions, can also result in test cases that do not represent real-world usage. (Mathur, 2008)

The outcomes of flawed test cases due to incorrect assumptions can be significant. If the expected software behavior is not matched, incorrect test results may often occur. This again can lead to false positives or negatives and as a result incorrect conclusions of the software's correctness are made. Additionally, flawed test cases may miss defects or vulnerabilities, especially in critical areas or edge cases influenced by incorrect assumptions. If undetected, these defects could cause problems in production environments. (Mustafa et al., 2021)

Addressing the impact of flawed test cases caused by incorrect assumptions involves a comprehensive strategy. It's crucial to have clear and precise requirements to develop accurate test cases. This requires collaboration among stakeholders to clarify any uncertainties and resolve misunderstandings. Validating test cases through user research, stakeholder input, and real-world testing helps to make sure that the assumptions align with actual user behavior and scenarios. To maintain the accuracy of test cases they should be regularly reviewed and refined according to changing requirements and feedback. Accuracy and reliability of testing can be improved by correcting the flawed test cases that result from incorrect assumptions. The quality and the integrity of the software can be improved by aligning the tests with the software's requirements and user expectations. This can lead to better customer satisfaction and user experience. (Mustafa et al., 2021)

2.4 Best Practices in Unit Testing

Unit testing becomes effective when certain key elements are in place. Firstly, automation is crucial. This means that tests are run automatically, following predefined steps, which ensures consistency and adherence to established principles. Secondly, the speed of test execution is essential. Tests should ideally complete within milliseconds to avoid unnecessary delays in the testing phase. Fast execution preserves the integrity of the feedback loop, ensuring that developers receive immediate insights into the code's behavior. As the number of unit tests increases within a system, the need for fast execution becomes even more important. Otherwise, the total time taken by tests could significantly hinder the usefulness of the testing process. Therefore, maintaining a balance between automation and rapid execution is vital for the efficacy of unit testing in the development process. (Acharya, 2014)

Unit tests should operate independently, without depending on the results of other tests or the order they are run in. This makes tests more resilient to changes in execution conditions, ensuring they reliably reflect system functionality. Likewise, separating tests from external factors like databases or files requires using substitutes, like test doubles, to create isolated and reproducible testing environments. (Acharya, 2014)

Unit testing is effective because it provides consistent and portable results over time and different locations. Tests should give the same results no matter when or where they are run, which helps ensure they accurately reflect how the system behaves. Additionally, it is important for tests to be clear and concise. They not only validate the system but also serve as documentation. This is evident in how tests are named, with descriptive names making test suites easier to understand. (Acharya, 2014)

Finally, it is crucial to note that unit tests are integral components of software. Just like how code is refined and improved through refactoring, it is necessary to continually update and enhance unit tests to maintain their effectiveness and manageability. Managing large test classes can be challenging, so breaking them down into smaller, more organized suites supports iterative development and makes them easier to handle. (Acharya, 2014)

Appel (2015) discusses the typical structure of a unit test through the following example. This approach is commonly known as "arrange, act, assert" or "build, operate, check," but here we follow Meszaros' (2007) terminology of setup, exercise, verify, and teardown, which is shown below in Code excerpt 1.

```
private final static int NEW_FETCH_COUNT = Timeline.DEFAULT_FETCH_COUNT + 1;

@Test public void setFetchCount() {
 // (1) setup (arrange, build) Timeline timeline = new Timeline();
 // (2) exercise (act, operate) timeline.setFetchCount( NEW_FETCH_COUNT ); }
 // (3) verify (assert, check) assertEquals( NEW_FETCH_COUNT, timeline.getFetchCount() );
}
```

Code excerpt 1. Meszaros' (2007) unit test structure

He states the following:

1. The first step sets up the object being examined, often called the system under test (SUT) (Meszaros, 2007). This establishes the starting point for the SUT, including specific inputs and conditions.

2. Once the setup is complete, attention turns to testing the particular functionality of the SUT. Usually, this means calling a single method and recording the results for evaluation.

3. Checking that the actual outcome matches what we expect follows. This involves verifying that the observed result aligns with the anticipated behavior.

4. It is important for a test to clean up after itself, returning the environment to its original state. This ensures that any changes made during setup or testing do not affect subsequent tests unexpectedly. Although teardown is not always included in basic unit tests, it is crucial for maintaining test independence and reliability.

In addition to a clear structure, it is important to take into consideration what inputs are used in the tests. Starting with the happy path, defined by Meszaros (2007) as the typical flow of actions within a software scenario, ensures smooth progress towards user or system objectives without unexpected issues. This prioritizes delivering the most important business value and aligns with what the component is expected to do. Focusing on the normal flow initially helps lay a solid foundation for meeting requirements and avoids potential inefficiencies that can come from dealing with edge cases too soon. Additionally, when a component has multiple functions, it is important to choose the happy path that brings the most business value. However, sometimes it is also helpful to begin with a simple function, known as a "low hanging fruit," especially in certain situations (Kaczanowski, 2013).

Although focusing on the happy path usually covers the main requirements of the component being tested, it does not mean the job is done. Often, the most critical issues come from boundary conditions, which might not show up until later in development but can have a big impact. It is important to find and address these less common cases through thorough testing strategies. Even with careful testing, there can still be oversights, which is where code coverage tools come in handy. They help identify areas of code that have not been tested yet and potential issues within the component. However, just because a path of code is covered does not mean it is thoroughly tested. Even a small change in a covered path could lead to unexpected results without causing existing tests to fail. (Appel, 2015)

Choosing the right names in software development is laborious. It is important for names to be clear and short, whether they are for classes, methods, or variables. Test names should describe exactly what is being tested, including what inputs or conditions are expected and what outcomes are anticipated. While naming patterns like `[UnitOfWork_StateUnderTest_ExpectedBehavior]` help with organization and clarity, other methods like using 'should' at the beginning aim to make test intentions clear. However, these approaches might make things more complicated, especially for complex behaviors. The challenge is finding a balance between making names meaningful and keeping them short and easy to read, as discussed by Appel (2015).

3 Natural Language Processing and LLM-based Software Engineering

In this study AI-powered LLMs are utilized in unit test generation. LLMs are based on NLP technology, which have been used in software engineering to create for example translation applications. In this chapter NLP technology is first described. Next, we move to discuss the LLM technology and its benefits and limitations.

3.1 Natural Language Processing Techniques in Software Engineering

NLP refers to the technology that converts speech or writing into a machine recognizable form. The technology matches the content on a semantic level, but it also has some common-sense knowledge and reasoning ability to overcome issues with context. (Cambria & White, 2014) NLP is one of the important fields of study in computational linguistics and AI research (Young et al., 2018) and has formed a basis for the development of Large Language Models.

The main aim of NLP is language translation, i.e. getting the computer to understand human language. Gurbuz, Rabhi & Demirors (2019) suggest that there are two main processes in the NLP mechanism. Natural language understanding (NLU) refers to processes where natural content is processed and recognized by the computer and natural language generation (NLG) refers to the process where the computer produces natural language. As mentioned, machine translation is one of the main aims of NLP. It involves both NLU and NLG. Advancements in speech recognition, machine translation technology and deep neural networks have shifted the research from written text translation to automatic translation of spoken language (Ghazizadeh & Zhu, 2020).

Second important research area in NLP is text categorization. This refers to the process of classifying documents into categories based on the content or attributes of the document (Dasgupta, 2009). Main area of research in this field is the construction of a classification model. Search engines also utilize NLP techniques (Taskin, 2019). The user's search query needs to be recognized, but also matching the search results to the query requires understanding of natural language.

There are many challenges in NLP, but one of the key difficulties is the ambiguity of natural language (Young et al., 2018). Ambiguity refers to homonyms, synonyms, context-dependency and other human language phenomena that change the meaning of an expression or give it multiple possible meanings. If this ambiguity is not properly handled, the computer cannot properly understand the expression. The computer also needs contextual information to interpret the expression correctly. The context may arise from surrounding sentences or environment. (Ghazizadeh & Zhu, 2020)

3.2 Overview of Large Language Models

LLMs are an advancement in NLP with their ability to process and generate natural language fluently and coherently. LLMs are based on transformer architectures which capture contextual information and relationships within textual data. They are trained on vast amounts of data that enables them to learn nuances and intricacies of language. LLMs have brought significant advances in machine translation, text summarization and text analysis. (Hou et al., 2024)

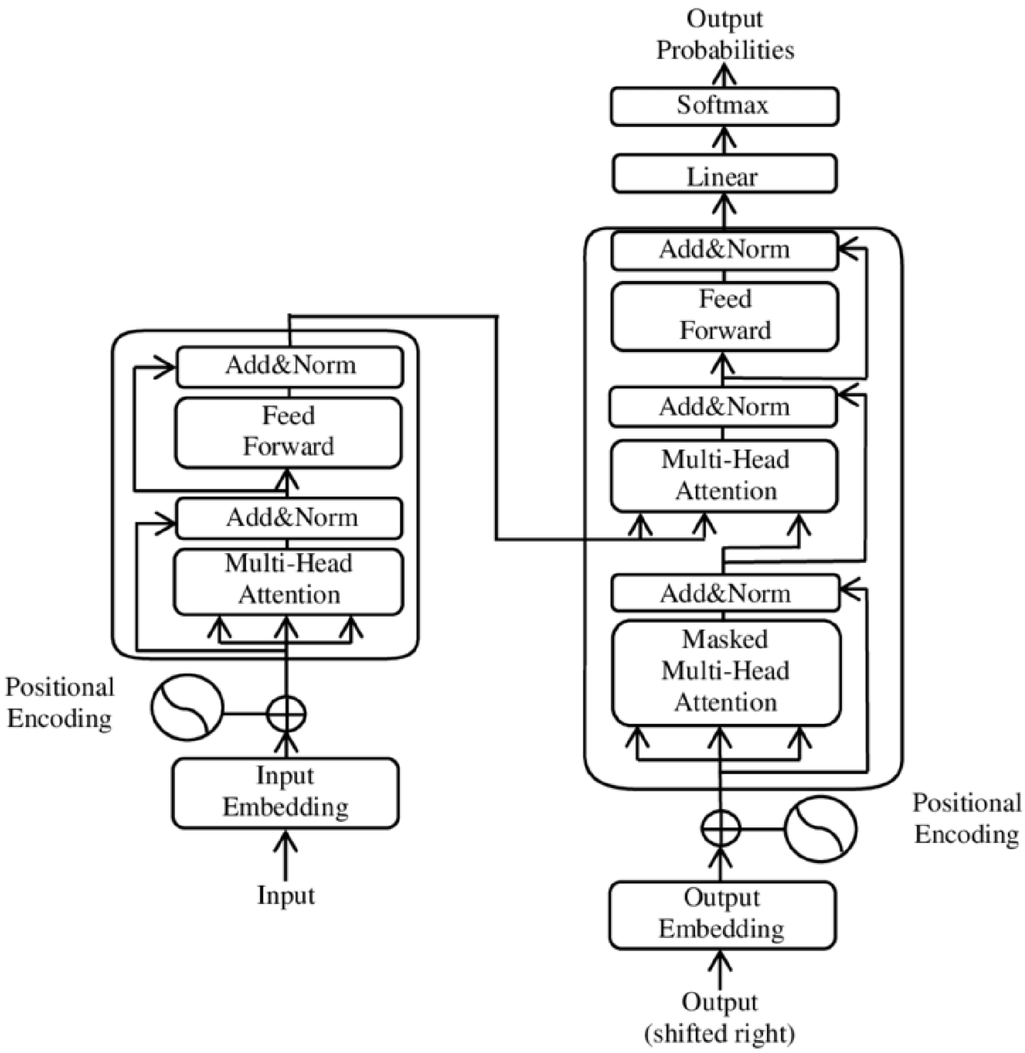

Figure 1. Main components of the transformer model from the original paper. (Wikipedia contributors[2])

The basis of LLMs is the transformer model, which extracts an encoder-decoder structure. Above in Figure 1 are the main components of the model. The encoder processes input and prepares it for a further decoding process, while the decoder produces output based on the representation of the encoder. The representation maintains the context information for each token of the input. These tokens interact with each other through self-attention, which refers to a mechanism to capture the relations and dependencies of the tokens. An attention score denotes an importance measure for each token, hence understanding long-range dependencies and context. The self-attention mechanism is operated many times in parallel, and with this, the notion of a multi-head attention model arises: each head handles different aspects of the input sequence. These are then combined to create the final representation of each token. As

[2] https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture), visited 10.6.2024

transformers do not intrinsically look at the order of the token sequences, LLMs add another dimensionality to the transformer, called positional encoding, which enriches information representing the position of all tokens in the input sequence to help the model discriminate between the tokens. (Vaswani et al., 2017)

Pre-training is an important phase in the process. During the process they are trained on extensive text data to learn general patterns of languages. This process is computationally intensive and involves many iterations. During training, the model's parameters are adapted so that the difference between the predicted and actual tokens in the training data is minimized. LLMs are generally pre-trained on huge amounts of text data collected from various sources: books, articles, websites, and many others. This data is cleaned and pre-processed to remove noise and ensure consistency. During pre-training, LLMs are trained to predict the next token in a sequence of text. The transformer architecture is used in the pre-training process, whereby the model can effectively capture long-distance dependencies and contextual information through constituent self-attention modules. (Naveed et al., 2024)

After pre-training on a large corpus of text data, LLMs can be fine-tuned for specific tasks or domains. This involves adjusting the model's parameters to improve performance on specific tasks. Fine-tuning uses transfer learning, where the knowledge gained from pre-training on a large dataset is applied to a new task with a smaller dataset. This allows LLMs to adapt to new tasks or domains more efficiently, requiring less labeled data for training. Fine-tuning usually involves giving the pre-trained LLM task-specific labeled data. The model's parameters are then adjusted using backpropagation based on the task's objective, like classification accuracy or sequence generation metrics. (Naveed et al., 2024)

LLMs are increasingly being utilized in various aspects of software development. They can be trained on large code repositories to generate code snippets, functions, or even entire programs. Tools like OpenAI's Codex and GitHub Copilot use natural language descriptions and partial code snippets to provide code suggestions and auto-completion. (Naveed et al., 2024)

LLMs can also be automated to generate for example documentation for codebases. They analyze the code comments, function signatures, and source code and based on the information they generate human-readable summaries describing the purpose, functionality, and usage of the code. (Bhattacharya et al., 2023) LLMs can also be used for static code analysis and bug detection. They are able to find potential errors, vulnerabilities, or bad practices in source code. LLMs are taught with large codebases and can spot common coding mistakes, suggest best practices, and also offer insights into code quality. (Venkatesh et al., 2024) Requirements engineering is also a part of LLMs repertoire as they can analyze natural language requirements documents, user stories, or feature requests and extract key insights, identify dependencies, and facilitate requirement extraction and prioritization (Hou et al., 2024).

LLMs can also aid in automated testing. They are able to generate test cases, identify edge cases, and predict potential failure scenarios based on natural language descriptions or specifications. This can complement traditional testing approaches and help ensure comprehensive test coverage, faster release cycles, and improved software reliability. (Wang et al., 2024) LLMs in test generation is discussed further in Chapter 5.

3.3 LLMs Capabilities and Limitations in Software Engineering

Large Language Models are promising tools in aiding software engineering. They have many capabilities and as the models are developed further even more can arise. However, currently there are still limitations to these capabilities that may eat away their efficiency. In this chapter current capabilities and limitations of LLMs in software engineering are discussed.

3.3.1 Capabilities

Large Language Models are showing proficiency in various areas of software engineering, including code generation, test generation, bug localization, verification, test automation, fault localization, program repair, code clone detection, code review, debugging, and bug reproduction. Hou et al. (2024) and Zhang et al. (2023) published large research review articles that discuss the current capabilities and limitations of LLMs in software engineering.

The articles' found that LLMs have demonstrated their ability to interpret natural language descriptions, code comments, and requirements, and then generate corresponding code snippets that fulfill the given specifications. This ability helps in quickly creating prototypes and automating repetitive coding tasks. LLMs are especially useful in program synthesis, improving productivity by generating code from high-level instructions. Their dual understanding of natural language and programming languages makes them suitable tools for advancing software engineering and streamlining the development lifecycle. The interactive coding approach, where code is run and the model receives feedback on its functionality, improves the model's ability to generate correct code. Overall, LLMs can help understand requirements and produce accurate code, speeding up development and improving software quality. (Hou et al., 2024, Zhang et al., 2023)

Using LLMs with software testing methods has improved test case generation, bug classification, and defect prediction, making the testing process more precise and efficient. For example, LLMs can be fine-tuned for specific projects to create custom test cases, helping to detect subtle bugs or security issues early. Additionally, combining LLMs with traditional static and dynamic program analysis enhances code analysis. LLMs use their natural language processing abilities to understand code-related text, such as comments and documentation, making it easier to grasp code functionality, identify dependencies, and produce relevant documentation. (Hou et al., 2024, Zhang et al., 2023)

LLMs like Codex have led to the development of commercial products such as GitHub Copilot and open-source models like StarCoder and Code LLAMA. These models, based on pretrained transformers, have been successful in code processing tasks. Research shows that LLMs can also use external tools for complex reasoning. For example, models like PAL and PoT extend Codex with Python interpreters for numerical calculations, while ViperGPT uses vision APIs to gather information from visual inputs. (Hou et al., 2024, Zhang et al., 2023)

3.3.2 Limitations

LLMs proved their potential in software engineering tasks, however they also come with several limitations. Hou et al. (2024) and Zhang et al. (2023) discuss limitations in their effectiveness, reliability, interpretability, privacy, and security.

In terms of effectiveness, LLMs have been examined in various coding tasks and their integration into development tools. They could enhance software maintenance and evolution, but there are challenges in using LLMs because they're large and need a lot of computing power. Also, training needs huge datasets that might introduce biases. There are concerns about how LLMs generate code ambiguously and if they work well across various tasks or areas. Moreover, training LLMs is costly, especially when fine-tuning them with specific data. In complex problems, LLMs might generate code solutions that aren't very effective. (Hou et al., 2024, Zhang et al., 2023)

Reliability is another limitation. To handle ambiguity in code generation, more context, domain-specific knowledge can be added, or multiple models used together. Making LLMs more adaptable to different software tasks is crucial for producing dependable code. (Hou et al., 2024, Zhang et al., 2023)

Interpretability is also a concern. The effectiveness of current evaluation metrics for judging LLM performance in software engineering is questioned, stressing the importance of interpretability and trustworthiness. Ethical concerns are also raised, especially about using LLM-generated code in real-world applications. Developers might be cautious about using LLM-generated code if they don't understand how it was generated. Tools must explain how the model works and why it produces certain outputs. (Hou et al., 2024, Zhang et al., 2023)

Security is also a limitation. The transparency of some models has been a concern, as many of the LLMs do not disclose how they are trained. The training data may have issues with quality, representativeness, and even ownership. There is a possibility of adversarial attacks, where the LLM is deliberately fed vulnerabilities. This can result in code suggestions that have exploitable weaknesses. Developers and stakeholders need to be aware of the issues like prompt injection attacks, source code vulnerability and data sensitivity and they should have strategies to address these issues. (Hou et al., 2024, Zhang et al., 2023)

LLMs like GitHub Copilot have raised concerns regarding privacy issues. These models are trained on vast amounts of data, including publicly available code from platforms like GitHub. Consequently, there's a chance of revealing sensitive or proprietary details from the training data. Also, LLMs can create code snippets that might accidentally contain confidential or copyrighted material. To use and deploy LLMs ethically privacy issues need to be considered and there needs to be measures to protect sensitive information. (Hou et al., 2024, Zhang et al., 2023)

Use of resources needs to be addressed also. LLMs are complex AI systems that require a considerable amount of computing power in both training and running. They often require specialized hardware, such as GPUs and TPUs. Energy consumption is also high, which raises environmental and financial issues. Plus, training LLMs needs vast amounts of data, which can be tricky to store and manage. (Hou et al., 2024, Zhang et al., 2023)

4 Unit Test Generation with Large Language Models

Large Language Models are adopted also in unit test generation. In recent years there have been many specified tools developed that are used in unit test generation. One such is GitHub Copilot's Gentest which is also used in the research conducted in this thesis. The field is naturally very young, therefore not a lot of research has yet been published. Next, we represent a short research review and then discuss further five studies focusing on unit test generation. The findings of these studies are discussed.

4.1 Research Review

Wang et al. (2024) published a systematic review of LLMs in software testing. They analyzed 102 relevant studies which used LLMs in testing. Focusing mainly on unit testing were 20 studies that were published between 2020-2023. Most of these (17 studies) were published in 2023 indicating that the research area is picking up, but also that there is a relatively small number of publications available. In evaluating the performance of unit test case generation across different studies, it's important to note that they use different datasets, making direct comparisons challenging. Main findings of these studies are discussed next.

Several studies have examined pre-training or fine-tuning LLMs for unit test case generation. This approach was commonly used due to the limitations of early-stage LLMs. Even in recent studies, this method persists, aiming to improve LLMs' understanding of domain knowledge. This approach proved efficient, and coverage and validity of the generated tests were significantly higher (Rao et al., 2023, Steenhoek et al., 2023, Shin et al., 2023, Tufano et al., 2020, Alagarsamy et al., 2023). Focus has then shifted towards designing effective prompts for LLMs to enhance their performance. Instead of relying solely on pre-training or fine-tuning, these studies aim to optimize LLMs by refining prompts to improve their understanding of context. Chen et al. (2023) proposed a generation-validation-repair mechanism that rectifies errors in generated unit tests. This method produced higher line coverage and their user analysis supported the mechanism as efficient. Similar post-generation-processing method was adapted by Dakhel et al. (2024) and Yuan et al. (2023). Both conclude that providing the LLM with feedback on the generated tests enhances the performance.

More innovative methods for test generation have been explored by Vikram et al. (2023). They propose the use of LLMs to generate property-based tests with the assistance of API documentation. They suggest that API method documentation can help LLMs create logic for generating random inputs and deriving meaningful result properties for verification. Similarly, Plein et al. (2023) generated tests based on bug reports from users, thus moving away from using source code to generate unit tests. Their results suggest that bug reports are a useful tool for the LLM as it is able to generate unit tests based on the reports.

Some studies have adopted an approach of using LLMs as support to traditional software testing techniques such as search-based techniques. Lemieux et al. (2023) used the LLM to further enhance coverage results after the selected traditional tool Pynguin plateaued in its efforts. Their findings show that this approach significantly enhances coverage results. Two studies have also been conducted to compare the LLMs performance against traditional search-based tools (Tang et al., 2024, Bhatia et al., 2023). Both found that the tests generated by LLMs were comparable to search-based software testing (SBST) techniques. Findings from these two studies are discussed in more detail in Chapter 4.2.

4.2 Related Work

In this section we discuss the findings of five recent studies on using LLMs in unit test generation. First, we represent a study that researched GitHub Copilot's abilities in Python code generation. Secondly, we discuss four studies that examined ChatGPT, GPT-3.5-Turbo, StarCoder and Codex. The study setups of these four studies are represented in more detail. Then the results of these studies are discussed, namely regarding code coverage, correctness and test smells. In Chapter 5.2.4 we discuss the results of a study using GitHub Copilot in test generation.

Yetistiren, Ozsoy et Tuzun (2022) published research that studied GitHub Copilot in code generation. They generated Python code using the HumanEval dataset. They extracted problems from the dataset and created both human solutions and GitHub Copilot generated solutions to the problems. Solutions generated by Copilot were then evaluated for their validity and correctness. This approach is similar to the current study's approach as validity is evaluated by the syntactical correctness of the code. However, correctness was evaluated against the human written unit tests. In this thesis we generate unit tests with Copilot and correctness is evaluated based on whether these generated tests pass. Yetistiren, Ozsoy et Tuzun (2022) reported a success rate of 91,5% in code validity. Correctness was distributed to three categories: correct, partially correct, and incorrect generations. One third of the generated code was correct and half of the generated code was partially correct. Approximately 20% was incorrect.

Bhatia et al., (2023) concluded a study comparing ChatGPT to Pyngyin in unit test generation. They examined large code samples that ranged from 100 to 300 lines of code. The main focus was on two types of code: 1) function-based modular code where functions are clearly defined, and they act like independent units of code and 2) class-based modular code where the primary units are structured around classes and objects. The prompts to generate the tests were designed using two parts: 1) a Python program (100-300 lines) and 2) a task description. ChatGPT was provided with the complete code to see if it can identify units. Next ChatGPT was prompted to "Write unit tests using Pytest for the given Python code that covers all edge cases." Then the generated tests were compared to those from Pynguin for statement and branch coverage, noting any missed statements. To improve coverage, a new prompt was created with the indices of missed statements, asking ChatGPT to generate more tests. This process was repeated until no further improvement was observed. The analysis was done by comparing the performance of unit tests generated by the two systems. Main focus was on statement and branch coverage across various code structures and complexities. After this the ChatGPT was iteratively prompted to enhance coverage until no longer any enhancement was achieved. Finally, the correctness of the generated tests was evaluated.

Siddiq et al. (2024) researched three LLMs (GPT-3.5-Turbo, StarCoder and Codex) on unit test generation. They extracted classes from open-source datasets SF110 (194 classes) and HumanEval (160). The LLMs performance was evaluated based on branch/line coverage, correctness and quality in terms of test smells. Second part of the study consists of evaluating how context influences the generated tests. They generated JUnit tests for scenarios that contained a different set of code elements and evaluated their performance based on compilation rates, code coverage, the number of correct unit tests, and the occurrence of test smells. Details of the test methodology were not discussed in the research paper.

Tang et al. (2024) conducted a comparative evaluation of LLMs (ChatGPT) and SBST in generating unit test suites. The generated suites were evaluated by their correctness, readability, code coverage and bug detection aiming to better understand the LLMs potential in unit test generation. The dataset used in the study consisted of 248 Java classes that were collected from 79 different projects. In investigating the bug detection ability of the LLM they used a dataset that contains 835 bugs from 17 projects. The test generation prompts were identified through a series of expressions: "Write a unit test for ${input}" with the code segment as input, "Can you create unit tests using JUnit for ${input}?" with the code segment as input, and "Create a full test with test cases for the following Java code: ${input}?" with the code segment as input. Based on these findings, the prompt was summarized as: "Write a JUnit test case to cover methods in the following code (one test case for each method): ${input}?" with the code segment as input. Their goal was not to compare and evaluate prompts to find the best-performing one, but to create a reasonable prompt that simulates how developers might use ChatGPT in a real-world environment.

Yuan et al. (2023) researched ChatGPT's abilities in unit test generation. They had a dataset of 1000 Java classes in executable project environments. Prompting the LLM was done by providing it with a natural language description of the task and a code context of the focal method. This contained the complete focal method, including the signature and body; the name of the focal class (i.e., the class that the focal method belongs to); the field in the focal class; and the signatures of all methods defined in the focal class. This was then completed with a natural language explanation as follows: “You are a professional who writes Java test methods. Please write a test method for the {focal method name} based on the given information using {Junit version}”. This procedure was applied to all focal methods. The generated tests were then evaluated for their correctness, coverage, readability, and usability. The two latter questions were studied through a user interview. Based on the findings of these research questions they also suggested a novel approach, called ChatTester, that generates unit tests via ChatGPT.

4.2.1 Coverage

Code coverage is used as an assessment tool in all the aforementioned studies. Both statement and branch coverage are employed. The results of statement coverage range between 93,26 - 55,4% and branch coverage 65,6 - 92,8%. Bhatia et al. (2023) achieved the highest percentages implying that their technique is the most efficient. Siddiq et al. (2024) reported results in line coverage. It cannot be directly assessed identical as statement coverage as line coverage entails the statements. Hence, the line coverage values of Siddiq et al. (2024) are not completely comparable to the other studies. All results are presented in Table 2.

Table 2. Code coverage percentages

	Bhatia et al.	Siddiq et al.	Yuan et al.	Tang et al.
Statement/Line coverage	91,55 - 93,26%	67 - 87,7%	82,3%	55,4%
Branch coverage	89,5 - 91,68%	69,3 - 92,8%	65,6%	NA

Bhatia et al. (2023) used iterative prompting in their research. Prompting was continued until the coverage results reached their peak values. Statement coverage was increased by 15,25 - 27,95% on average. Tang et al. (2024) on the other hand created a "reasonable prompt", as they describe it. This refers to finding the prompt through a series of trials and then using the same prompt derived from this to create test cases. Yuan et al. (2023) provided the LLM with a natural language description and the code context in the prompt but did not iteratively enhance the prompt. This suggests that using a template-based prompt is not an effective style. Providing the LLM with more information of the context clearly enhances the coverage reached and iterating the prompts yields the best coverage results. Tang et al. (2024) note that the incomplete specifications and lack of feedback mechanism may have contributed to the low coverage values in their research.

4.2.2 Correctness

Correctness is used to evaluate unit tests. The term may refer to compilation and failing assertions and the nature of the errors in failing assertions. Siddiq et al. (2024) report the percentage of compilable unit tests of four different LLMs (Codex (2K), Codex (4K), GPT3.5-Turbo, and StarCoder) when the LLM is not provided context. StarCoder performed the best with 70% compilable tests, whereas the other LLMs generated less than half compilable tests. Tang et al. (2024) found similarly that 69,6% of the tests generated by ChatGPT were compilable and Yuan et al. (2024) had 42,1% compilable tests. However, when the LLMs were provided with context, the compilation rate increased to up to 53,8% (Siddiq et al., 2024). Also, a notable change was GPT-3.5-Turbo's performance, which dropped to 2.5%. This was found to result from duplicated package declarations.

Siddiq et al. (2024) observed that there were repeatedly similar syntax errors that caused the compilation errors when the LLM was not provided with the context. They found that LLMs 1) tend to create additional test classes that may not be fully developed, 2) often include explanations in natural language before and after the generated code, 3) information is sometimes repeated such as the class being tested or the test prompt, 4) package declaration is changed or even removed, 5) generated integer constants that exceed the usual maximum value and 6) incomplete unit tests resulted when the test code reaches a certain limit. They then applied automated heuristics to fix these issues and were able to improve the compilation percentage to 76,9 - 100%. Siddiq et al. (2024) noted that tests that could not be fixed through heuristics were found to contain semantic errors, such as unknown symbols, incompatible conversions and abstract class instantiations. Unknown symbols were the most common error type. Yuan et al. (2023) similarly report frequent occurrences of unresolved symbols, type errors, access errors and invalid instantiation of abstract classes.

Tang et al. (2024) also found that compilation errors are often a result of ChatGPT's attempt to predict parameters, parameter types and such. They argue that this is due to the fact that the LLM did not have an overview of the entire project. Siddiq's (2024) findings support this as the results were improved (without the exception of GPT-3.5-Turbo) after providing the LLM with context. Yuan et al. (2024) suggest that a post-generation validation mechanism could improve the compilation percentage.

The amount of passing test methods were also analyzed to evaluate the tests correctness. Bhatia et al. (2024) reported that 39% of Category 2 (function-based modular code) and 28% of Category 3 (class-based modular code) assertions were incorrect in the tests generated. They argue that ChatGPT may generate correct assertions better to a well-defined structure within the code. Yuan et al. (2023) similarly reported that only 24,8% of the generated tests were executed without any execution errors. Majority of these execution errors (85,5%) were assertion errors. All of these errors were found to be caused by incorrect assertions generated by ChatGPT. Rest of the execution errors were different types of runtime exceptions, which according to Yuan et al. (2023) may imply that ChatGPT may be unaware of external resources during test generation.

In their analysis Siddiq et al. (2024) examined the correctness by categorizing the results into two categories: 1) all methods pass, i.e. correct tests and 2) some methods pass, i.e. somewhat correct tests. The two different datasets (HumanEval and SF110) yielded different results. The

LLMs generated 52,3 - 81,3% correct tests with the HumanEval dataset, whereas the SF110 performed poorly with 6,9 - 51,9% correct tests. GPT-3.5-Turbo performed the worst, generating the lowest scores and StarCoder had the best results. Somewhat correct tests were reported to be 81,3 - 92,3% of the HumanEval dataset and 16,1 - 62,7% of the SF110 dataset. In this category GPT-3.5-Turbo performed best with the HumanEval dataset and surprisingly the worst with the SF110 dataset. Results were similar in their second scenario where the LLM was provided context. Overall, the correctness results of these studies imply that LLMs often fail to comprehend the focal method, thus generating low quality tests. However, as Siddiq et al. (2024) point out, the somewhat correct tests are also useful.

4.2.3 Test Smells

The quality of unit tests can also be measured by the presence of so-called test smells. Steenhoek et al. (2023) found that unit tests generated by LLMs often contain test smells. Unit tests generated by OpenAI Codex Cushman model were analyzed to find four different test smell types. They found that Duplicate Asserts and Conditional Test Logic were commonly observed, but Redundant Print and Empty Test were rare. They noted that the tests containing the smells were complex, hence challenging to comprehend.

Table 3. Test smells in the HumanEval dataset found by Siddiq et al. (2024)

Test Type	Frequency
Assertion Roulette	Common
Conditional Logic	Common
Empty Test	Common
Exception Handling	Common
Eager Test	Common
Lazy Test	Most common
Duplicate Assert	Common
Unknown Test	Common
Magic Number Test	Most common

Siddiq et al. (2024) conducted more thorough research on the occurrence of test smells in LLM generated unit tests. They analyzed the tests on 16 different test smell types and found 9 of them occurring. Based on the analysis the LLMs generate in the HumanEval dataset Assertion Roulette, Conditional Logic Test, Empty Test, Exception Handling, Eager Test, Lazy Test, Duplicate Assert, Unknown Test and Magic Number Test types. Magic Number Test and Lazy Test were the most frequent. Assertion Roulette, Eager Test and Duplicate Assert were also common. These are presented in Table 3. In the SF110 dataset Magic Number Test, Assertion Roulette and Eager Test types are most common. Additional test smells were also observed, which did not occur in the HumanEval dataset. These were Constructor Initialization, Mystery Guest, Redundant Print, Redundant Assertion, Sensitive Equality, Ignored Test and Resource Optimism. These findings are presented in Table 4. The results in both datasets were similar in the scenario of providing the LLM with context.

Table 4. Test smells in the SF110 dataset found by Siddiq et al. (2024)

Test Type	Frequency
Magic Number Test	Most common
Assertion Roulette	Most common
Eager Test	Most common
Constructor Initialization	Occasional
Mystery Guest	Occasional
Redundant Print	Occasional
Redundant Assertion	Occasional
Sensitive Equality	Occasional
Ignored Test	Occasional
Resource Optimism	Occasional

4.2.4 Research on Test Generation with GitHub Copilot

GitHub Copilot is an AI-powered code completion tool that assists developers in writing code by suggesting completions based on context[3]. It integrates with existing IDEs, such as Visual Studio Code and IntelliJ. Copilot utilizes OpenAI's Codex LLM, which has been fine-tuned on open-source GitHub projects. One of its notable features is test generation, where it can generate tests for code snippets, aiding in creating comprehensive test suites. This feature is particularly useful for scenarios where existing tests are lacking, as it can generate tests even without prior test cases. The generated tests can be influenced by varying the code and comments in their code files before invoking Copilot, prompting questions about the optimal formulation of code comments to improve test generation usability.

[3] https://github.com/features/copilot, visited 10.6.2024

GitHub Copilot's test generation performance was studied by El Haji et al. (2024). The study investigates the effectiveness of generated tests with and without an existing test suite, placing particular emphasis on scenarios lacking pre-existing tests. It examines how manipulating code and comments before using Copilot influences test generation, prompting questions about the best way to formulate code comments for improved test usability. The research specifically evaluates the usability of test generations under different test method comment strategies, both within and without an existing test suite. The generated tests were assessed on their syntactic correctness, runtime correctness, passing and coverage.

In initial manual assessments, they noted that method comments affect tests produced by Copilot. To investigate further, tests were categorized into two groups: those with comments and those without. They explored four types of method comments. The first type, called "Minimal Method Comment," provides a brief description of a test method. In contrast, the "Behavior-Driven Development Comment" describes a scenario using the format "Given x when y then z." Another type, the "Usage Example Comment," includes a code snippet demonstrating a potential call of the code under test (CUT), without necessarily linking to a specific test scenario. Lastly, they examined the effectiveness of combining all these comment styles into a single method comment, a "Combined comment". These different approaches aimed to understand how comment styles influence the usability and quality of Copilot-generated tests.

The results of El Haji et al.'s (2024) study shows similar results as studies discussed previously. They found that 54,72% of tests with-context and 92,45% without-context failed. Most common reasons for tests to fail were syntax and runtime errors (22,64% with-context, 71,70% without-context). Most common runtime errors were non-existent attributes and incorrect parameters. Thus, they argue that Copilot does not consider the CUT, it relies only on other test methods in its context. However, the closed-source nature of Copilot does not allow us to examine the exact context used to prompt test generations. Also, 16,98% of tests with-context and 18,87% without-context failed due to assert mismatch, which leads to conclude that Copilot did not have sufficient information to determine the expected value of one or multiple assertions. Additionally, the amount of passing tests were 45,28% with-context and 7,55% without-context. El Haji et al. (2024) that some of the passing tests appeared to mimic other tests in the context. Without the context Copilot cannot mimic the context, hence the lower rate of passing tests.

When examining passing tests With-Context, it was observed that out of 24 generated passing tests, 17 covered the same branches as their human-written counterparts. Additionally, one generated test covered the same branches and even more new branches. Only six generated tests covered strictly fewer branches or included new branches. Consequently, the majority of passing tests generated by Copilot With-Context do not cover fewer branches than the original test, which positively affects usability. However, when considering tests generated Without-Context, only one test covered the same branches as its human-written counterpart, while the rest covered fewer or new branches. This suggests that tests generated Without-Context, even if passing, are less suitable. Overall, El Haji et al. (2024) conclude that the usability of Copilot-generated tests is poor as most tests need to be modified.

In their examination of the effect of commenting the test method, El Haji et al. (2024) finds that using the Usage Example Comment strategy resulted in the highest percentage of passing tests (34.78%) and the lowest number of broken tests (17.39%) within an existing test suite context. Additionally, this strategy produced test generations with the highest average ratio of covered branches matching their human-written counterparts. Similarly, they observed that employing the Combined Comment strategy resulted in the highest percentage of passing tests (21.74%) and the lowest number of broken tests (30.43%) in situations where there was no existing test suite. Additionally, this approach generated test iterations with the highest average ratio of covered branches matching those in the human-written tests.

5 Research Settings and Methods

In this chapter we describe the current study's definition and methodology. A description of the system under testing is also given. After this the results are presented. The results of research question 1 and 2 are discussed together in Chapter 5.4.1 and questions 3 and 4 are discussed individually in chapters 5.4.4 and 5.4.3.

5.1 System Description

The system, initially developed for a single research institute in 2013, is now being expanded to two additional large research institutes. It is essentially a project management platform that supports various functionalities such as project planning, solicitation of statements, and processing of applications and permits. Automation in specific areas improves efficiency and serves collaboration, facilitating the sharing of research findings. The system also provides a robust and user-friendly digital working platform for researchers.

The system is designed for scalability, and it provides services to organizations of different sizes and supports a range of roles, including researchers, research coordinators, and project coordinators. It is initially set to support about 400 users and manage 1300 to 1500 applications and permits. The system complies with national legislation and rigorous information system architecture requirements, with interoperable interfaces for seamless integration. Its modular design allows for the integration of existing features and flexibility for future needs.

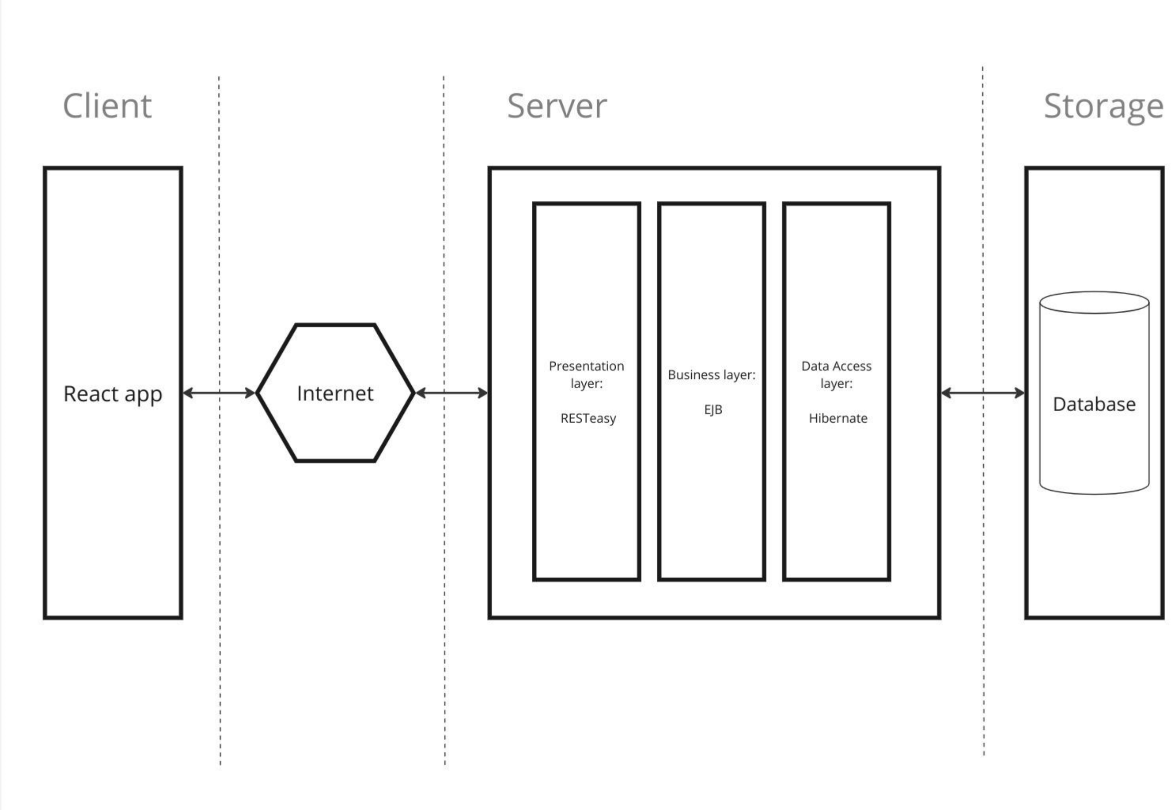

Figure 2. System architecture

The system employs Enterprise Java Beans (EJB) technology for database connections and has subsequently integrated representational state transfer (REST) architecture as shown in Figure 2. The backend technology has been kept constant, but the REST interfaces were implemented later to enhance the system's functionality. In its current version, the connections between the frontend and backend are entirely managed through REST calls, ensuring efficient communication and data exchange. This architectural approach allows for more flexible and scalable interactions between the system components. The EJB container in the application server is vital, providing a runtime environment for enterprise beans and managing services such as transaction management, security, and remote access. This enables the system to handle complex business logic and maintain transactional integrity, crucial for research management and digital services (Jendrock et al., 2013).

5.2 Research Questions

RQ1: How valid are GitHub Copilot's test suggestions in terms of compilation errors?

This research question seeks to evaluate the validity of GitHub Copilot's test suggestions by focusing on their ability to compile successfully. The investigation involves analyzing the suggested test cases to determine the frequency and types of compilation errors that occur. Compilation errors may result from syntax errors, type mismatches, unresolved references and other similar issues that prevent the code from compiling. The goal is to quantify the proportion of test suggestions that compile without errors and identify common patterns or specific areas where GitHub Copilot struggles. By understanding the validity in terms of compilation, we can assess the immediate usability of Copilot's test suggestions in a software development workflow.

RQ2: How correct are GitHub Copilot's test suggestions in terms of execution errors?

This research question aims to assess the correctness of GitHub Copilot's test suggestions by examining the occurrence of execution errors. Execution errors refer to runtime issues such as exceptions, incorrect logic, or failures in the test cases when they are run against the code. The study involves running the generated tests and recording the types and frequencies of execution errors. This includes analyzing whether the tests correctly validate the intended functionality and whether they produce false positives or negatives. By evaluating the correctness in terms of execution, the research seeks to determine how reliable and useful GitHub Copilot's test suggestions are in practice.

RQ3: How effective are GitHub Copilot's test suggestions in terms of code coverage?

This research question investigates the effectiveness of GitHub Copilot's test suggestions by measuring code coverage. Code coverage refers to the extent to which the test cases exercise the different parts of the codebase, including statements and branches in our research. The research involves generating tests with GitHub Copilot, running them, and using coverage analysis tool JaCoCo to measure the resulting code coverage metrics. The aim is to find out whether testing is comprehensive by identifying which areas of the code are well-tested and which are not. High code coverage implies thorough testing, which leads to better code quality as bugs are caught.

RQ4: Are there specific test smell trends in GitHub Copilot's test suggestions?

This research question aims to identify and analyze the presence of test smells in GitHub Copilot's test suggestions. Test smells are indicators of potential problems in test code that can lead to maintenance issues or unreliable tests. Common test smells include excessive setup code, hard-coded values, lack of assertions, or overly complex test logic. The study involves systematically reviewing the generated test cases to detect recurring patterns of test smells. By identifying these trends, the research seeks to understand the quality and maintainability of the tests suggested by GitHub Copilot. This analysis can reveal areas where GitHub Copilot's test generation might need improvement to produce better quality tests.

5.3 Methodology

The research methodology employs a comprehensive one-shot method to evaluate the performance of GitHub Copilot in generating and refining test cases. This approach is structured into three primary steps, ensuring a thorough assessment of GitHub Copilot's capabilities. The test setup is described in Figure 3 below.

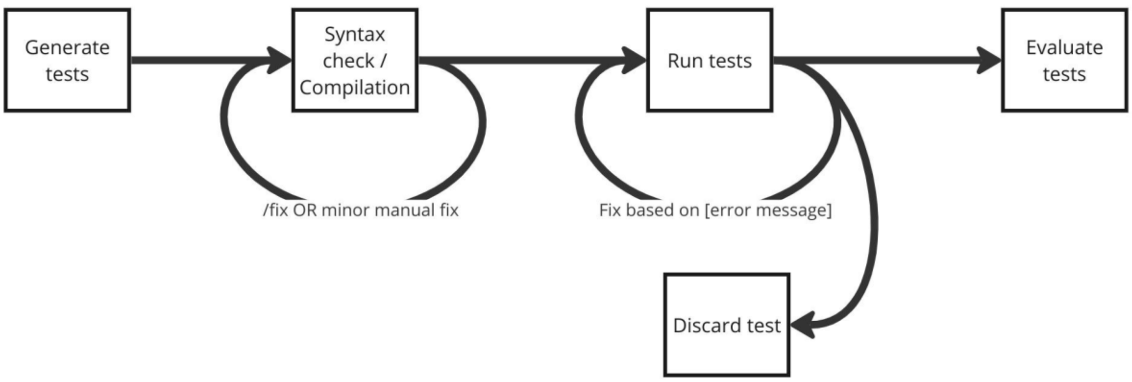

Figure 3. Test setup

Step 1: Test generation

The initial phase involves prompting GitHub Copilot to create a test case. This process begins by selecting a method within the integrated development environment (IDE) using the right mouse button, which opens a contextual menu. Within this menu, the "Generate tests" option under the GitHub Copilot section is selected. Copilot's response, containing the generated test cases, appears in the chat box. These test cases are then carefully extracted from the chat box and saved into a designated test class for further evaluation.

Step 2: Syntactic correctness verification and correction

Following the generation of test cases, their syntactic correctness is verified using IDE hints and feedback mechanisms. If a test case is syntactically correct, it proceeds directly to the execution phase. However, if the test case contains syntax errors, the methodology involves using GitHub Copilot's "Fix this" function to request corrections. Copilot provides suggested corrections in the chat box, which are then extracted and saved into the test class.

At this stage, the viability of Copilot's suggested corrections is critically assessed. If the corrections are significantly off-target or require extensive manual intervention to become functional, they may be rejected. Conversely, if the corrections are close to being accurate and only require minimal manual adjustments, these adjustments are made manually. This step ensures that only potentially viable test cases move forward in the process.

Step 3: Test execution and quality evaluation

Once the test cases are confirmed to be syntactically correct, they are executed to verify their functional correctness. Successful execution of a test case indicates that it is ready for the quality evaluation phase. If a test case fails during execution, the system-generated error messages are provided to GitHub Copilot with a request for further corrections, articulated using natural language commands in the chat box. The responses from Copilot, containing the suggested fixes, are reviewed, extracted, and incorporated into the test class. The test is then re-executed to check for successful execution. This cycle of requesting corrections and re-execution is repeated a few times to explore Copilot's ability to rectify errors.

If, after multiple attempts, the test case continues to fail and the necessary corrections are not easily identifiable or feasible, the test case is ultimately rejected. However, if a successful correction is achieved, the test case is executed again to ensure its correctness and is subsequently moved to the quality evaluation phase. During quality evaluation, various metrics such as code coverage and the presence of test smells are assessed to determine the overall quality and maintainability of the test case.

To evaluate code coverage values are extracted from JaCoCo. It operates by first instrumenting the code, which involves adding extra code to Java classes to monitor coverage.

This can be done either at runtime with a Java agent or before runtime during the build process. As tests are run, the instrumented code collects execution data, showing which parts of the code were executed. After testing, JaCoCo creates reports in formats like HTML, XML, or CSV, which help developers see how much of their code is covered by tests and spot any areas that need more testing.

Analysis of the presence of test smells is done with tsDetect, a tool that is designed to identify test smells in the code. The test smell detector works by first identifying the test and production files in a project. It then parses these files to create an Abstract Syntax Tree (AST). Each test smell detection module checks the AST for specific issues based on set rules. For example, to find "Redundant Print" smells, the tool looks at method calls in test methods to spot unnecessary print statements. The results are saved in a CSV file, showing whether each smell is present or not. (Aljedaani et al., 2021) These results are utilized in our analysis.

This methodology provides a reasonable understanding of GitHub Copilot's ability to generate valid, correct, and high-quality test cases in a system that utilizes the Java EE system by documenting key metrics. The methodology assesses Copilot's current capabilities and identifies specific areas for potential improvement, contributing insights into the development and refinement of automated test generation tools.

5.4 Results

In this chapter we represent the results of our research. Research questions 1 and 2 (validity and correctness) are discussed together. After this code coverage results are considered and lastly, test smells are evaluated. All chapters include examples of the tests generated by GitHub Copilot.

5.4.1 Validity and Correctness

GitHub Copilot generated a total of 62 tests of which 18 had to be discarded. 44 tests were accepted either one shot valid or correct or after corrections. Only 6 tests needed validity corrections, whereas correctness proved harder for GitHub Copilot and 21 tests needed to be corrected. For one focal method Copilot generated 1-3 test methods. Copilot was commonly able to generate all test methods one shot valid and correct if the focal method had more than one test method. This would suggest that test generation is more effortless when Copilot decides to generate more than one test method. Results of validity and correctness are presented in Table 5.

Table 5. Validity and correctness of generated tests.
Classes named “E+number” are EJB classes and names that end in “R” are API classes.

Focal method		Test method	One shot valid	One shot correct
E1	getEntity1FromDTO	getEntity1FromDtoTest	x	
E1	getByEntity1Ids	shouldReturnEntity1ListWhenEntity1IdsProvided	x	x
		shouldReturnEmptyListWhenNoEntity1IdsProvided	x	x
		shouldReturnEmptyListWhenEntity1NotInCache	x	x
E1	getEntity8AsEntity1	shouldReturnEntity1WhenEntity8Exists	x	
		shouldCreateNewEntity1WhenEntity8DoesNotExist	x	
E1	getEntity1ByIdentifier	shouldReturnEntity1WhenIdentifierExists	x	
		shouldThrowExceptionWhenMultipleEntity1WithSameIdentifier	x	
		shouldReturnNullWhenIdentifierDoesNotExist	x	
E1	cloneEntity1	shouldCloneEntity1Successfully	x	x
E2	getPopulatedListBySearch	getPopulatedListBySearchReturnsCorrectList	x	x
		getPopulatedListBySearchReturnsEmptyListWhenNoMatch	x	x
E2	createEntity2	createEntity2ReturnsCorrectlyPopulatedObject	x	
		createEntity2ReturnsNewObjectWithUniqueIdentifier	x	x
		createEntity2AssignsCorrectTila	x	
E2	populateEntity9	populateEntity9AssignsCorrectValuesWhenEntity9DoesNotExist	x	
E2	populateEntity10	testPopulateEntity10		
E2	populateEntity3	testPopulateEntity3	x	x
E2	createEntity2FromDTO	testCreateEntity2FromDTO	x	x
E2	getEntity11	testGetEntity2Tilatiedot	x	x
E3	createEntity3	shouldCreateEntity3ForExternalUser	x	x
E3	createEntity3FromDTO	shouldCreateEntity3FromValidDTO	x	x
E3	copyEntity3	shouldCopyEntity3Successfully		
E3	hasRightToEdit	shouldReturnResponseWhenGetEntity2ByProjectIdIsCalled	x	
E4	getValueByIdAndType	getValueByIdAndTypeReturnsCorrectValue		

E4	getValuesByType	getValuesByTypeReturnsCorrectValues	x	x
		getValuesByTypeReturnsEmptyArrayWhenNoMatch	x	x
E12	createEntity12	testCreateEntity12	x	
E5	createEntity5	shouldCreateEntity5Successfully	x	x
E5	findByParameter1AndParameter2	shouldFindByParameter1AndParameter2Successfully	x	x
		shouldReturnEmptyListWhenNoEntity5Found	x	x
E5	createEntityFromDTO	shouldCreateEntityFromDTONewEntity	x	x
		shouldCreateEntityFromDTOExistingEntity	x	x
E6	createEntityFromDTO	createEntityFromDTOTest	x	
E2R	getEntity2ListByEntity3Id	shouldReturnPermissionErrorWhenUserHasNoPermissionForProjectId	x	
E2R	getEntity2ById	shouldReturnServerErrorWhenExceptionOccurs	x	
E2R	getEntity2ListForUser	shouldReturnEntity2ListForUser	x	
		shouldHandleExceptionWhenGettingEntity2ListForUser		
E2R	getParameter3Entity2ListForUser	shouldReturnParameter3Entity2ListForUser	x	
		shouldHandleExceptionWhenGettingParameter3Entity2ListForUser		
E5R	getEntity5List	shouldReturnAllEntity5WhenGetIsCalled	x	x
		shouldReturnEmptyListWhenNoEntity5Exist	x	x
E5R	saveNewEntity5	testSaveNewEntity5		x
E5R	deleteEntity5	testDeleteEntity5	x	x

Copilot was asked to fix errors that were either due to syntactical or execution errors. The prompt contained the error message and a natural language prompt “Correct test based on this error message”. As stated earlier 18 of the generated tests could not be corrected through this procedure. In these cases, Copilot generated such poor quality test code that manual corrections would have replaced almost entirely the generated test. As these would not anymore represent Copilot’s abilities, the tests were discarded.

There were several issues due to which the generated tests were discarded. Firstly, there were misconfigurations in the mock objects, which were ineffective for the test scenario. Secondly, Copilot regularly creates unnecessary mocks and at the same time leaves out crucial mocks. This resulted repeatedly in null pointer exceptions during the execution of the tests. When a test was affected with both of the issues simultaneously, the resulting test code had only one or two usable lines. Manually correcting these would have taken considerable effort, hence they were discarded. An example of a discarded test is represented in Code excerpt 2.

```java
@Test
void shouldReturnEntityWhenIdExists() {
    Long existingId = 1L;
    Version version = new Version(1L);
    Entity13 expectedEntity = new Entity13();
    when(entity13DAO.findById(existingId, version)).thenReturn(expectedEntity);

    Entity13 result = entity13EJB.getById(existingId, version);

    assertEquals(expectedEntity, result);
}
```

Code excerpt 2. Discarded test

On some occasions a test had to be rejected due to Copilot not being able to detect an obvious error. In Figure 4 Copilot has generated a test with two methods named identically. It did not spot this on its own, even after a correction request was made. Copilot's answer of not being able to detect any problems demonstrates it lacking abilities in detecting issues that to the human eye seems very straightforward.

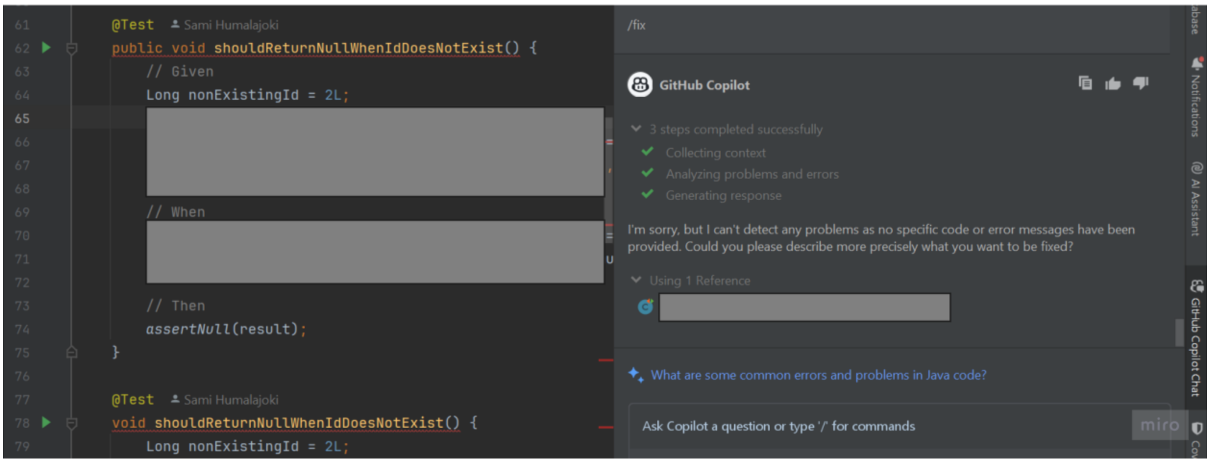

Figure 4. Screenshot of Copilot's correction suggestions

Another demonstration of Copilot’s limitations on understanding contextual issues is represented below in Code excerpt 3 and 4. In its first suggestion Copilot uses a wrong Mockito method, “`.returns`”. After the error message was provided to Copilot, it was able to successfully correct the issue and used the ".thenReturn" method. A possible reason behind this behavior of using non-existent methods can be that Copilot misunderstands or oversights the context it is provided. It is possible that Copilot did not comprehend the correct usage of the Mockito framework resulting in a hallucinated method.

```
@Test
public void shouldReturnEmptyListWhenSearchDoesNotMatch() {
    Entity13Search search = new Entity13Search();
    Version version = new Version(1L);
    when(entity13DAO.findBySearch(search, version)).returns(new ArrayList<>());
    List<Entity13> result = entity13EJB.getPopulatedListBySearch(search, version);
    assertTrue(result.isEmpty());
}
```

Code excerpt 3. First suggestion by Copilot

```
@Test
public void shouldReturnEmptyListWhenSearchDoesNotMatch() {
    Entity13Search search = new Entity13Search();
    Version version = new Version(1L);
    when(entity13DAO.findBySearch(search, version)).thenReturn(new ArrayList<>());
    List<Entity13> result = entity13EJB.getPopulatedListBySearch(search, version);
    assertTrue(result.isEmpty());
}
```

Code excerpt 4. Successful correction after fix-prompt

The Code excerpts 5 (Initial suggestion), 6 (Correction prompt) and 7 (Corrected suggestion) below demonstrate another mocking issue in Copilot's tests. In the first suggestion it did not mock `entity8`. The error message was provided to Copilot and the corrected suggestion had a properly mocked `entity8`, which prevented the initial execution error.

```
    @Test
    public void shouldReturnPermissionErrorWhenUserHasNoPermision() {
        // Given
        Long id = 1L;
        when(Entity3EJB.hasRightToEdit(anyLong(), any())).thenReturn(false);

        // When
        Response response = entity2Resources.getEntity2ListByEntity3Id(id);

        // Then
        verify(entity3EJB, times(1)).hasRightToEdit(anyLong(), any());
    }
```

Code excerpt 5. First suggestion by Copilot

```
Correct test based on this error message:

[ERROR] Tests run: 2, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 0.461 s <<<
FAILURE! - in package1.Entity2ResourcesTest
[ERROR] package1.Entity2ResourcesTest.shouldReturnPermissionErrorWhenUserHasNoPermission
Time elapsed: 0.305 s  <<< ERROR!
java.lang.NullPointerException: Cannot invoke "package2.Entity8.getUserName()" because
"entity8" is null
        at package1.BaseRestResources.checkPermission(BaseRestResources.java:103)
        at package1.Entity2Resources.getEntity2ListByEntity3Id(Entity2Resources.java:164)
        at
package1.Entity2ResourcesTest.shouldReturnPermissionErrorWhenUserHasNoPermission(Entity2R
esourcesTest.java:59)
```

Code excerpt 6. Prompt given to Copilot to fix the issue

```java
@Test
public void shouldReturnPermissionErrorWhenUserHasNoPermission() {
    // Given
    Long id = 1L;
    Entity8 entity8 = mock(Entity8.class);
    when(userSessionEJB.getEntity8()).thenReturn(entity8);
    when(entity3EJB.hasRightToEdit(anyLong(), any())).thenReturn(false);

    // When
    Response response = entity2Resources.getEntity2ListByEntity3Id(id);

    // Then
    verify(entity2EJB, times(1)).hasRightToEdit(anyLong(), any());
}
```

Code excerpt 7. Correction suggestion by Copilot

All in all, Copilot does not perform consistently. There is no clear pattern to be found behind the incorrect behavior. Mocking appears to be the main stumbling block. One possible reason behind this erratic behavior is the fact that Copilot is not able to test its suggestions. It generates the suggestions based on its training data and understanding of programming patterns, but it has no internal feedback system. Without this capability to execute and validate the code, it may produce incorrect suggestions. This may be especially true in complex scenarios where context is pivotal, mocking being one of these scenarios. Copilot relies very much on the prompt and the context it's provided, so ambiguities in either can lead it astray. Thus, Copilot still requires human oversight and validation to generate good quality and correct output.

5.4.2 Code Coverage

Analysis of the code coverage results reveal that Copilot's performance is again inconsistent. The line coverage values of the generated tests vary between 0% and 100%. Similarly, branch coverage results alter between 0% and 100%. All values are represented in Table 6. The table also includes Cyclomatic complexity values that measure the different independent paths through a program's source code, showing its complexity and the variety of ways it can run. The number of lines within the class is also shown in the table.

Table 6. Code coverage values of generated tests

Element	Line coverage	Branch coverage	Cyclomatic complexity	Lines
getEntity1FromDTO	89%	50%	13	34
getByEntity1Ids	100%	100%	3	8
getEntity8AsEntity1	81%	40%	6	25
getEntity1ByIdentifier	100%	100%	3	9
cloneEntity1	100%	100%	2	21
getPopulatedListBySearch	100%	100%	2	8
createEntity2	100%	50%	2	15
populateEntity9	89%	50%	5	16
populateEntity10	100%	75%	3	12
getEntity11	0%	0%	4	7
createEntity3	92%	50%	2	18
createEntity3FromDTO	89%	56%	107	367
copyEntity3	85%	31%	9	97
getValueByIdAndType	94%	50%	2	7
getValuesByType	100%	n/a	1	4
createEntity12	100%	n/a	1	5
createEntity5	100%	n/a	1	2
findByParameter1AndParameter2	100%	50%	2	6
createEntityFromDTO	64%	20%	6	19
createEntityFromDTO	84%	25%	7	44
getEntity2ListById	24%	21%	8	34
getEntity2ByEntity3Id	36%	25%	5	19
getEntity2ListForUser	74%	33%	4	17
getParameter3Entity2ForUser	77%	40%	6	19
deleteEntity5	66%	50%	3	12
getEntity5List	100%	n/a	1	6
saveNewEntity5	75%	50%	2	12
Average values	80%	50%		

Certain patterns can be found when analyzing these results. If a test has a 100% line and branch coverage, it generally has a low cyclomatic complexity and low number of lines. These test scenarios are straightforward, single-path methods. On the other hand, high line coverage and lower branch coverage is found in tests that have a moderate cyclomatic complexity and a varying number of lines. This may suggest that some branches are missed even though they cover multiple paths. Further on, a high cyclomatic complexity and significant number of lines combined with low line and branch coverage, imply insufficient test coverage in complex methods. A method with high cyclomatic complexity typically has many conditional branches, loops or other control flow structures. This naturally creates more potential execution paths that need to be covered. There are also some cases where line coverage is 100% and branch coverage n/a. These have a very low cyclomatic complexity and low number of lines, which indicates that there are no branches, hence no branch coverage value available.

On the other end of the spectrum are tests that have a 0% coverage, in either of the two values or both. This indicates that the method or some paths are not tested at all. Code excerpt 8 represents one of these situations. In this example the focal method itself is mocked by Copilot and not executed. Also, unnecessary mocks are added.

```
@Test
void testGetEntity11() {
    // Arrange
    Entity8 mockedEntity8 = Mockito.mock(Entity8.class);
    when(entity14EJB.getEntity8()).thenReturn(mockedEntity8);
    when(mockedEntity8.getEntity8TechnicalId()).thenReturn(1L);

    Entity2 mockedEntity2 = Mockito.mock(Entity2.class);
    Entity11DTO realEntity11DTO = new Entity11DTO();
    doReturn(realEntity11DTO).when(entity2EJB).getEntity11(any());

    // Act
    Entity11DTO result = entity2EJB.getEntity11(mockedEntity2);

    // Assert
    assertEquals(realEntity11DTO, result);
}
```

Code excerpt 8. Spectacular failure

Copilot typically isolates the test method to minimize any dependencies, which might lead to this type of behavior. This ensures that tests can run independent of external factors, as is a normal testing strategy, but as is seen in this example, it may lead to excess mocking. However, this incorrect mocking may also result from complexity of the method. By mocking the method itself, Copilot might try to avoid the intricacies of the method. In this case the end result is an oversimplification that results in no test being performed.

In conclusion, the more complex the method is, the lower coverage it seems to reach. A correlation between high cyclomatic complexity and lower line coverage is found, but not with branch coverage. When the complexity value is under 4, line coverage is on average

95%, but branch coverage is 71%. On the other hand, complexity value over 3 results in average line coverage of 72% and branch coverage of 36%. It needs to be stated that the research data has only two methods with a significantly high cyclomatic complexity, so understanding this tendency would need more data. But it is clear based on the results that the more complex the code is, the more difficult it is to achieve complete, or even higher, coverage to Copilot. The results are not surprising; when the focal method is practically always called only once with a single input per test, it can only follow one path per test.

5.4.3 Test Smells

Results from the test smell analysis of the generated tests are presented in Table 7. The table contains only those categories of which an occurrence was found. The tsDetect tool has found 54 different tests. The difference in the number of tests can be explained through the difference in interpretation of test methods. The tool also includes the initialization methods in the overall number. The following percentages are calculated on previously mentioned 44 tests.

Table 7. Test smells in generated tests

Test class	Number of methods	Assertion Roulette	Exception Catching Throwing	Print Statement	Eager Test	Lazy Test	Unknown Test	Magic Number Test
E1	11	0	0	0	0	0	0	0
E2	11	2	0	0	4	8	2	6
E3	4	0	0	0	0	0	0	0
E4	4	1	0	0	0	2	0	3
E5	2	0	0	0	0	0	0	0
E12	6	0	0	0	0	4	0	5
E6	2	0	0	0	0	0	0	0
ER1	9	2	1	1	0	8	2	8
ER2	5	0	0	0	0	0	0	0
Sum	54	5	1	1	4	22	4	22

Most common types of test smells found were Magic Number Test and Lazy Test. Both had 22 occurrences, which means that 50% of the tests had either type of test smell. Magic Number Test is a test that uses hard-coded values in the test code, whereas Lazy Test performs weakly and might miss important validations. Assertion Roulette (5 occurrences), Eager Test (4 occurrences) and Unknown Test (4 occurrences) follow with significantly lower prevalence to Magic Number Test and Lazy Test. Assertion Roulette refers to a situation where the test has multiple assertions, but there is no clear indication which of them failed. Eager Test on the other hand results from situations of testing too much at once, making the demonstration of failures harder. Unknown Tests on the other hand are tests that potentially point to unstructured or poorly defined test cases as they do not fit into any recognizable patterns.

The significantly higher number of certain tests over others may indicate various factors influencing GitHub Copilot's behavior. Firstly, Copilot might be biased to prioritize certain patterns or behaviors. Since it is a closed system, it is not possible to investigate this further. But also, the complexity of the code and code coverage may play a role here. The more complex the code and functionalities are, the higher the occurrence of certain test smells, for instance Magic Number Test and Lazy Test, may be.

Copilot seems not to apply best practices of unit test generation, which results in the high prevalence Magic Number Tests. This may be a result of Copilot focusing on readability or maintainability of the tests, leading it to overlook coding standards. The instances of Magic Number Tests did not occur in edge cases or less common scenarios that many times make human developers overlook best practices. Code excerpt 9 exhibits a Magic Number Test smell. In the test Copilot has mocked a method making it return an arbitrary number "1L", which has no specific meaning. The method where the number is inputted could also be mocked itself. This technique does not provide any additional value to this test.

```
@Test
void testPopulateEntity3() {
    // Arrange
    Entity2 entity2 = new Entity2();
    Entity3 mockedEntity3 = Mockito.mock(Entity3.class);
    when(mockedEntity3.getId()).thenReturn(1L);
    when(entity3EJB.getById(anyLong(), any())).thenReturn(mockedEntity3);

    // Act
    entity2EJB.populateEntity3(entity2);

    // Assert
    assertEquals(mockedEntity3, entity2.getEntity3());
}
```

Code excerpt 9. Magic number test smell example

GitHub Copilot is marketed as increasing productivity and taking over mundane tasks, so that human developers can focus on more creative tasks. This strongly implies that speed is emphasized when optimizing its performance. Pursuing quick results when prompted may lead to Copilot oversimplifying complex scenarios. This again can lead to generation of simplistic tests, which only verify the most obvious or direct outcomes. It might not even have a goal of thoroughly going through the code and its various aspects. This is seen in a high prevalence of Lazy Test Smells. A more detailed prompt might lead to more thorough testing, but as this research's goal was to replicate a real-world use case of GitHub Copilot, complex and detailed prompts are not in the scope of this study.

6 Conclusions

This case study evaluated unit tests generated by GitHub Copilot by their validity, correctness, coverage, and quality based on test smells. Overall, 62 test cases were generated of which 44 were included in the study. The discarded 18 tests were not executable and correcting these issues would have required considerable effort, in some cases rewriting the test fully. The study provides insights into the capabilities and limitations of GitHub Copilot with test generation automation. The findings of each research question are shortly summarized. Then implications of the findings are discussed and lastly a short discussion of future research goals concludes the study.

6.1 Summary of Findings

In this chapter the findings of each research question are summarized. First, the findings on research questions 1 and 2, validity and correctness, are discussed. Next findings for research question 3, code coverage, are briefly explained. And lastly, summarization of research question 4, test smells, is given.

6.1.1 Validity and Correctness

The first research question (RQ1) examined Copilot’s ability to generate compilable unit tests. Out of the 44 tests selected for this study only 6 required validity corrections. This suggests that Copilot is able to produce valid tests, however there were suggestions that failed to compile. These failures were mainly due to mock object configurations and the generation of unnecessary mocks or mocks completely missing. This suggests that Copilot has significant limitations in handling complex mocking scenarios without human intervention.

The second research question (RQ2) focused on the correctness of the tests by analyzing the occurrence of execution errors. The study found this to be more challenging to Copilot than validity. Out of the 44 accepted tests, 21 required corrections due to failures. Common correctness issues were runtime exceptions, incorrect logic, or failures in validation of functionalities. Performance was also inconsistent where some tests were generated correct in the first attempt, while others required several correction rounds. The results show that Copilot has issues in detecting obvious errors, e.g. duplicate method names. It also often uses incorrectly the mocking frameworks.

In conclusion, Copilot is able to generate valid and correct unit tests, but the performance is inconsistent. Human intervention is needed frequently. The primary areas of difficulty for Copilot include handling complex mocking scenarios, detecting straightforward errors, and maintaining contextual understanding during test generation. The lack of an internal feedback system to execute and validate the generated code contributes to these limitations. Copilot is not able to test its own code, and therefore it relies heavily on the provided context. This in return may lead to hallucinated methods and other errors that risks the reliability of the generated tests.

6.1.2 Code Coverage

Research question 4 (RQ4) aimed to evaluate the effectiveness of Copilot's tests by the means of code coverage. Coverage analysis tool JaCoCo was utilized in measuring line and branch coverage values. JaCoCo also provides measurement of cyclomatic complexity of the CUT's complexity, and this was utilized in the analysis. The results highlight a significant variability in Copilot's performance. Both coverage values ranged from 0% to 100%. When these inconsistent values are evaluated against the cyclomatic complexity values of the code, we found that the more complex the focal method is, the more Copilot struggles.

The results show a clear inverse relationship with cyclomatic complexity and code coverage. Both line and branch coverage values were higher when the focal method's cyclomatic complexity value was low. For example, methods with a cyclomatic complexity under 4 had an average line coverage of 95% and branch coverage of 71%. In contrast, methods with a cyclomatic complexity over 3 had significantly lower coverage, with an average line coverage of 72% and branch coverage of 36%.

The complex methods that proved difficult for Copilot often had many conditional branches and control flow structures, which created several execution paths. Covering these comprehensively with test automation by Copilot proved challenging. Reversely, with simple, single-path methods Copilot performed well. It often achieved 100% line and branch coverage in methods with low cyclomatic complexity and low number of lines. These results indicate that Copilot is very effective in straightforward testing scenarios.

An additional challenge for Copilot was mocking issues, in this case excess mocking. Its tendency to minimize dependencies led in some situations to excess mocking. This resulted in not executing the actual method, which in turn resulted in 0% coverage. This behavior was again observed in complex methods, which implies that Copilot might oversimplify the test to avoid intricate dependencies, which leads to failing to perform meaningful tests.

In conclusion, Copilot generates comprehensive test cases in simple focal methods, but struggles with more complex methods. And as was observed in research question 1 and 2, mocking is challenging to Copilot and the challenges are bigger the more complex the focal method is.

6.1.3 Test Smells

Lastly, research question 4 evaluated the quality of Copilot's tests by the prevalence of test smells. The aim was to study whether recurring patterns were found and provide insights to Copilot's limitations. Our findings suggest that certain types of test smells occur more often in Copilot's suggestions than others and code complexity is also at play here. Copilot's behavior also suggests biases towards certain patterns, and it may prefer speedy response time over good quality. There also seems to be challenges in adhering to test generation best practices.

The most common test smell types were Magic Number Test and Lazy Test. These occurred in more complex focal methods. This suggests that complexity affects Copilot's performance in test generation. The more complex the code and functionalities were, the more these test smell types occurred. The results indicate that Copilot prefers certain patterns and behaviors, such as introducing magic numbers or creating multiple, similar tests for one focal method. This might lead to think that the system is biased inherently to utilize these types of unwanted patterns and behaviors. It may also favor speed over quality, in expense of thorough and robust test generation. GitHub markets the system as increasing productivity and might be pursued by emphasizing speedy response times of the system. This may lead to the generation of simplistic tests that verify only the most obvious or direct outcomes.

Results of the fourth research question also indicate that Copilot does not apply best practices of unit testing very consistently. The high amount of Magic Number Tests and Lazy Tests is a testament of Copilot's tendency to overlook coding standards. It may do this in order to favor readability and maintainability of the tests. The root cause of this behavior may lay in the emphasis of quick and straightforward suggestions.

All in all, Copilot produces tests of reasonable quality, but it is affected by the complexity of the code and its possible, inherent features. Complex code may lead it to overlook coding standards and start using undesirable patterns and behavior. It may also favor speed over quality, which in return weakens the test code quality.

6.2 Future Research

Future research on GitHub Copilot's capabilities is needed. Most evident issue is the handling of complex test scenarios that involve intricate mocking. It could be beneficial to investigate how Copilot could be provided with internal feedback. It might be beneficial if it could execute and validate its own suggestions and thus enhance its reliability.

This study was performed with a relatively small dataset. In the future exploring a larger and more diverse dataset could provide a more comprehensive understanding of Copilot's effectiveness. Using methods with varying complexities might yield more insights on the reasons behind the challenges Copilot faces in complex code. It could also provide understanding to the relationship between complexity and code coverage. Adding other metrics could also be beneficial. In addition to the metrics used in this study, test execution time, fault detection capability and maintainability of the tests could provide more insights to the tool's performance.

In this study prompting was repeated similarly in all test generation situations. More detailed and specific prompts might produce better results. However, writing intricate prompts is time consuming and resource demanding, which could simply change the nature of the developers work, but not provide any benefits in resources. So, even though this approach might optimize the performance of the tool, it may not bring any real-life benefits. Lastly, future research could benefit from examining the generated tests longitudinally. Assessing their maintainability and reliability over time could provide insights to the long-term impact of this type of automated test generation.

6.3 Implications

GitHub Copilot shows potential in assisting the unit test generation process. It creates well-structured and clearly named tests that provide savings in resources. That being said, its abilities diminish in situations involving complex code. It is very effective in generating tests for methods with low cyclomatic complexity, and in such cases performs very productively. Copilot's challenges with mocking need to be considered when using the tool. Even though tests are generated, they may be meaningless and oversimplified. As it is also prone to using unwanted patterns, such as test smells, the suggestions need to be carefully reviewed. Reviewing and refining the generated tests may prove hard to automate as Copilot is on some occasions oblivious to the most obvious errors in its suggestions.

This study underscores the need for human insight in the test generation process. Copilot can streamline the process and provide useful suggestions, but human developers must always, even for the simple code, review the suggestions. GitHub Copilot can serve as a helping hand, but it cannot yet replace the expertise and critical judgment of developers. In conclusion, while GitHub Copilot shows promise in generating effective tests for simple methods, its performance is inconsistent and particularly challenged by complex methods. Developers can utilize Copilot for straightforward scenarios but should be prepared to review and refine its suggestions in more complex codebases.

References

Acharya, Sujoy. Mastering unit testing using Mockito and JUnit. Packt Publishing Ltd, 2014.

Alagarsamy, Saranya, Chakkrit Tantithamthavorn, and Aldeida Aleti. "A3test: Assertion-augmented automated test case generation." arXiv preprint arXiv:2302.10352 (2023).

Aljedaani, Wajdi, et al. "Test smell detection tools: A systematic mapping study." Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering. 2021.

Appel, Frank. Testing with JUnit. Packt Publishing Ltd, 2015.

Beck, Kent. Test driven development: By example. Addison-Wesley Professional, 2022.

Bhatia, Shreya, et al. "Unit test generation using generative ai: A comparative performance analysis of autogeneration tools." arXiv preprint arXiv:2312.10622 (2023).

Bhattacharya, Paheli, et al. "Exploring Large Language Models for Code Explanation." arXiv preprint arXiv:2310.16673 (2023).

Cai, Xia, and Michael R. Lyu. "The effect of code coverage on fault detection under different testing profiles." Proceedings of the 1st International Workshop on Advances in Model-based Testing. 2005.

Cambria, Erik, and Bebo White. "Jumping NLP curves: A review of natural language processing research." IEEE Computational intelligence magazine 9.2 (2014): 48-57.

Chekam, Thierry Titcheu, et al. "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption." 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017.

Chen, Yinghao, et al. "ChatUniTest: A Framework for LLM-Based Test Generation." arXiv e-prints (2023): arXiv-2305.

Cornett, S. “Code Coverage Analysis” Electronic source (1996), https://www.bullseye.com/coverage.html, visited 10.6.2024

Dakhel, Arghavan Moradi, et al. "Effective test generation using pre-trained large language models and mutation testing." Information and Software Technology 171 (2024): 107468.

Dasgupta, Sajib, and Vincent Ng. "Mine the easy, classify the hard: a semi-supervised approach to automatic sentiment classification." Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP. 2009.

Van Deursen, Arie, et al. "Refactoring test code." Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001). Citeseer, 2001.

El Haji, Khalid, Carolin Brandt, and Andy Zaidman. "Using GitHub Copilot for Test Generation in Python: An Empirical Study." (2024).

Garousi, Vahid, and Barış Küçük. "Smells in software test code: A survey of knowledge in industry and academia." Journal of systems and software 138 (2018): 52-81.

Ghazizadeh, Eghbal, and Pengxiang Zhu. "A systematic literature review of natural language processing: Current state, challenges and risks." Proceedings of the future technologies conference. Cham: Springer International Publishing, 2020.

Gurbuz, Ozge, Fethi Rabhi, and Onur Demirors. "Process ontology development using natural language processing: a multiple case study." Business Process Management Journal 25.6 (2019): 1208-1227.

Hemmati, Hadi. "How effective are code coverage criteria?." 2015 IEEE International Conference on Software Quality, Reliability and Security. IEEE, 2015.

Hou, Xinyi, et al. Large Language Models for Software Engineering: A Systematic Literature Review. 2024.

Jay, Caroline, Jorge Arturo Wong-Mozqueda, and Robert Haines. "Is code quality related to test coverage?." International Workshop on Sustainable Software Systems Engineering. 2015.

Jendrock, Eric, et al. The Java EE 6 tutorial: advanced topics. Addison-Wesley, 2013.

Kaczanowski, Tomek. Practical unit testing with JUnit and mockito. Tomasz Kaczanowski, 2013.

Kim, Dong Jae, Tse-Hsun Chen, and Jinqiu Yang. "The secret life of test smells-an empirical study on test smell evolution and maintenance." Empirical Software Engineering 26 (2021): 1-47.

Lemieux, Caroline, et al. "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models." 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023.

Madeyski, Lech. Test-Driven Development: An Empirical Evaluation of Agile Practice. 1. Aufl. Berlin, Heidelberg: Springer-Verlag, 2010. Web.

Mathur, Aditya P. "Foundations of software testing." Dorling Kindersley 184 (2009).

Meszaros, Gerard. xUnit test patterns: Refactoring test code. Pearson Education, 2007.

Mustafa, Ahmad, et al. "Automated test case generation from requirements: A systematic literature review." Computers, Materials and Continua 67.2 (2021): 1819-1833.

Naveed, Humza, et al. "A comprehensive overview of large language models." arXiv preprint arXiv:2307.06435 (2023).

Panichella, Annibale, et al. "Test smells 20 years later: detectability, validity, and reliability." Empirical Software Engineering 27.7 (2022): 170.

Plein, Laura, et al. "Automatic generation of test cases based on bug reports: a feasibility study with large language models." Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings. 2024.

Rao, Nikitha, et al. "CAT-LM training language models on aligned code and tests." 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023.

Saleh, Hazem. JavaScript Unit Testing. Packt Pub., 2013.

Shin, Jiho, et al. "Domain Adaptation for Deep Unit Test Case Generation." arXiv e-prints (2023): arXiv-2308.

Siddiq, Mohammed Latif, et al. "Using Large Language Models to Generate JUnit Tests: An Empirical Study." (2024).

Steenhoek, Benjamin, et al. "Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation." arXiv preprint arXiv:2310.02368 (2023).

Tang, Yutian, et al. "Chatgpt vs sbst: A comparative assessment of unit test suite generation." IEEE Transactions on Software Engineering (2024).

Tosun, Ayse, et al. "On the effectiveness of unit tests in test-driven development." Proceedings of the 2018 International Conference on Software and System Process. 2018.

Tufano, Michele, et al. "An empirical investigation into the nature of test smells." Proceedings of the 31st IEEE/ACM international conference on automated software engineering. 2016.

Tufano, Michele, et al. "Unit test case generation with transformers and focal context." arXiv preprint arXiv:2009.05617 (2020).

Turhan, Burak, et al. "How effective is test-driven development." Making Software: What Really Works, and Why We Believe It (2010): 207-217.

Venkatesh, Ashwin Prasad Shivarpatna, et al. "The Emergence of Large Language Models in Static Analysis: A First Look through Micro-Benchmarks." arXiv preprint arXiv:2402.17679 (2024).

Vikram, Vasudev, Caroline Lemieux, and Rohan Padhye. "Can large language models write good property-based tests?." arXiv preprint arXiv:2307.04346 (2023).

Wang, Junjie, et al. "Software testing with large language models: Survey, landscape, and vision." IEEE Transactions on Software Engineering (2024).

Wei, Yi, Bertrand Meyer, and Manuel Oriol. "Is branch coverage a good measure of testing effectiveness?." Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures (2012): 194-212.

Yetistiren, Burak, Isik Ozsoy, and Eray Tuzun. "Assessing the quality of GitHub copilot's code generation." Proceedings of the 18th international conference on predictive models and data analytics in software engineering. 2022.

Young, Tom, et al. "Recent trends in deep learning based natural language processing." ieee Computational intelligenCe magazine 13.3 (2018): 55-75.

Yuan, Zhiqiang, et al. "No more manual tests? evaluating and improving chatgpt for unit test generation." arXiv preprint arXiv:2305.04207 (2023).

Zhang, Ziyin, et al. "Unifying the perspectives of nlp and software engineering: A survey on language models for code." arXiv preprint arXiv:2311.07989 (2023).