Computer Fraud and Security
ISSN (online): 1873-7056

AI-Augmented Testing: GitHub Copilot for JUnit/Mockito Generation

Sandeep Reddy Gundla

Lead Software Engineer, MACYS Inc, GA, USA

gundlasr@gmail.com

Received: 20 July, 2025 ***Accepted: 27 September, 2025*** ***Published: 27 October, 2025***

Abstract

In recent years, AI has reached a revolutionary stage of impact in the software testing sphere, particularly in the context of unit test generation using AI. One of the brightest examples of this trend is that of GitHub Copilot. This is an artificial intelligence-based tool using machine learning and natural language processing that can automatically generate JUnit and Mockito test cases based on the current code's overview. The questionnaire of meaning checks the capacity of Copilot to replace the traditional testing methods with the creation of precise and exhaustive tests, resulting in a simultaneous increase in the productivity of the developers. The research compares AI-generated and human-written test suites based on various open-source Java projects by taking into consideration such key performance metrics as code coverage, the time taken for execution, and what in a language should be the defect detection. Empirical evidence shows that AI-generated tests cover 75% of the code base, which is greater than manually written tests 60%, It is said to take 40% less time to write a test. However, Copilot is found to be weak in terms of complex business logic and handling the cases of boundary conditions, as well as signaling the need for human developers to rectify them afterward, incorporating active transformation services. The results are promising for the effectiveness of AI-based instruments to enhance the speed of the testing process. However, they still emphasize that human intervention is imperative to ensure the quality and integrity of the products of the test generation. This paper can augment existing literature examining the topic of artificial intelligence in software testing and reinforce the fundamental idea that AI-enhanced tools can transform testing processes in significant ways, creating long-term value for both developers and the software industry as a whole.

Keywords: *GitHub Copilot, AI-driven test generation, JUnit, Mockito, Software testing efficiency.*

1. Introduction

Artificial Intelligence (AI) has become an integral part of software engineering, particularly in the area of automated testing. The history of using AI in software development dates back to the mid-2000s, with the advent of machine learning models and natural language processing, which aim to increase developer productivity. The use of AI has grown beyond code completion, bug fixing, and refactoring to test generation and optimization; however, early uses of AI were mainly in code completion, bug fixing, and refactoring. In this context, AI models have been trained to codebase analyze and generate test cases automatically, which are crucial in ensuring software reliability and performance. JUnit and Mockito have been used for unit testing in Java-based software development for a long time.

JUnit has been around since the late 1990s and has changed the testing landscape in that time, allowing a simple but effective testing framework to be used and understood for the creation of repeatable tests. Later, Mockito entered the scene to fill the demand to mock dependencies during unit tests, and became one of the reasons why developers could write clean tests that did not require dealing with complex or outside systems. These frameworks were instrumental during testing, but writing unit tests is still a tedious and error-prone process. Artificial Intelligence-powered tools, such as GitHub Copilot, have emerged as transformers, enhancing traditional testing practices by providing recommendations for code completion, test case generation, and even mock generation for testing purposes.

The developers can take advantage of the introduction of unit testing frameworks such as JUnit and Mockito, but they still encounter severe problems in writing good unit tests. The system logic is also complicated, and the codebase is quite large. In that case, manual testing requires a greater understanding of the system's logic, which takes a considerable amount of time. Other tests may also have limited quality, for instance, by not covering edge cases or incorrectly mocking external dependencies. To meet the needs of parameterization and to hasten the rate of software delivery, testing is reduced, which leads to software defects and maintenance management challenges. These hurdles can be reduced using tools like GitHub Copilot, an AI-driven tool. By utilizing machine learning and natural language processing, AI can assist in generating unit tests that are very accurate and carry a high coverage. Copilot will know what the code is trying to do and will then be able to create test cases, mocks, and assertions automatically. However, the impact and relevance of these AI tools in software testing are still an ongoing area of research.

The main research question is whether GitHub Copilot, as applied directly by an AI system, has a meaningful impact on the productivity and quality of test generation for JUnit and Mockito. This work assesses the case for the quickness, accuracy, and completeness of Copilot in generating test coverage relative to manually written tests.

Furthermore, the research will also seek to determine the degree to which the AI-generated tests are as robust and transparent as those that experienced developers create. This thesis is dedicated to the comparison of manually written unit tests and AI-generated unit tests obtained from GitHub Copilot. By comparing both approaches for several codebases, the research will determine the efficiency and effectiveness of AI-generated tests. The scope will also include the practical integration of AI tools into the existing software development process, with emphasis on the effect on productivity and test quality in the real world.

The article is divided into different chapters. This literature study aims to analyze the research results available on the topic of AI in software testing, with a focus on unit testing and test generation tools. The methodology chapter will consist of how the work on AI-driven test generation is done (GitHub Copilot will be described in detail). The results chapter will present a statistical analysis of the experiment, which was conducted to compare AI-written and manually written tests. The results of the discussion will be analyzed, and the paper will have a part dedicated to the future work and possible enhancements to AI testing solutions. The conclusion will summarize the significant findings of the study and the implications of the findings.

2. Literature Review

2.1 The Rise of AI in Software Engineering

Artificial Intelligence (AI) has undergone a revolution in various software engineering activities, including software coding, debugging, and testing. The juxtaposition of machine learning codes and models of natural language processing has paved the way for an evolutionary shift in the development tool of software, which will further improve the productivity and efficiency of software developers themselves [33]. The AIs can now provide the options of auto proposed code completions, fault detection, and, in some instances, even provide functions that can lead to considerable savings in coding time and also plug the mistakes that people commit. Researchers in AI for testing are now utilizing tools to automatically conduct unit, integration, and performance tests, thereby reducing the testing process. Figure 1 below illustrates the evolution of AI in software engineering, from AI assistants that assist with code completion and chat (Level 1) to fully autonomous (Level 5) agents, which can design, test, and debug code independently. This development shows that AI is becoming more vital in the automation of software development, enhancing productivity, and simplifying the testing process with various solutions like GitHub Copilot.

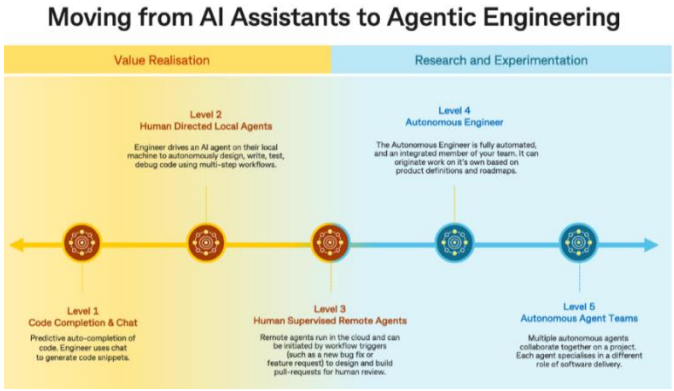

Figure 1: Progression from AI assistants to fully autonomous engineering teams in software development

A typical example supporting the rise of AIs in software testing is the application of machine learning algorithms to analyze written code and automatically initiate tests to ensure the software is corrected. Artificial intelligence used in software testing is also effective in identifying common bugs, suggesting test scenarios, and emulating real-world conditions, thereby significantly reducing human effort. According to empirical studies, AI-based tools are effective in automating repetitive testing processes, improving test accuracy, and enhancing code coverage. Testing is also faster using these tools, allowing developers to focus more effort on the higher-level design tasks and hence decreasing the total duration of the software product, bringing it to the market [14].

2.2 Unit Testing and Test Automation

Unit testing is recognized as one of the most critical tasks of the software development life cycle, and it is the assurance of the functional integrity of the individual pieces in a software system. The leading cause of unit tests is to ship code that works under different and varied conditions. In the past, developers had to manually create tests, which was time-consuming and prone to human error. JUnit and Mockito are unit testing frameworks that serve as a basis for tools to help ease this process [13]. As a popular unit-testing framework for Java-based applications, JUnit offers a relatively straightforward programming interface for creating small and repeatable test suites. Mockito also provides a gateway to me AC in the form of simulating external dependencies, as a way for software testing programs, thus allowing you to isolate the software being tested [6].

The direction of automation of tests has been driven by the fact that it has tried to offer a higher level of testing efficiency and requires consistency. Creating tests manually is slow by nature and is almost always unable to cover all the edge cases. To overcome these limitations, automated test generation, for example, made possible through tools like GitHub Copilot, aims at generating tests in accordance with the changing code through the application of AI. Not only does it reduce the time required to conduct the tests, but it also increases the likelihood of identifying defects early in the development cycle. Empirical studies imply that test automation can increase the software production speed by over 50 percent and, at the same time, decrease software defect cases in production.

2.3 GitHub Copilot and AI in Code Generation

GitHub Copilot, an AI code generation technology, has become a landmark in the application of AI to code generation. GitHub (in collaboration with OpenAI) Science Copilot uses large language models trained on large brand Corpora of published code to assist code writers in creating small code pieces, functions, and entire methods [38]. The tool has a high potential to improve the automation of routine codes and improve the general development cycle. As shown in Figure 2 below, GitHub Copilot assists in creating code and test files using machine learning models. The tool offers real-time code completion, where, based on simple prompts, service and test files are auto-generated for developers. This reduces the development cycle, especially if you are doing repetitive things such as writing unit tests. Copilot helps generate code snippets and test cases, enhancing efficiency and making it a powerful tool for software engineers seeking to automate code generation and testing processes.

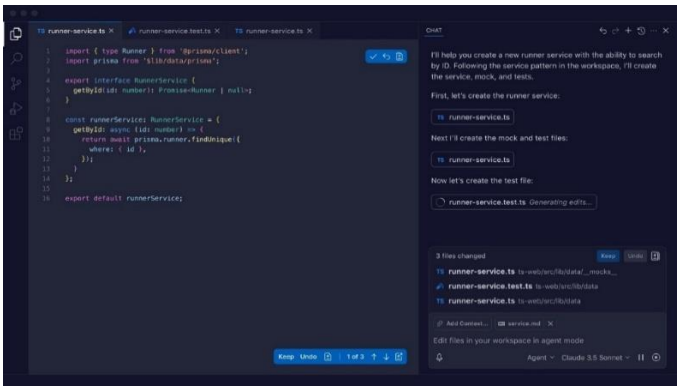

Figure 2: GitHub Copilot generating code and test files for efficient development automation

Studies have highlighted the effectiveness of Copilot in programming activities, ranging from test generation. Copilot will create unit tests automatically for frameworks like JUnit or Mockito and thus simplify the testing process. The efficiency of Copilot is particularly notable in scenarios where edge testing is required, as it can create a wide range of test scenarios with only moderate developer involvement. However, a concern is the precision of the code generated, which poses risks in a complex use case where code generation should be fine-tuned expertly and strictly to verify the authenticity of the tests.

The performance of Copilot depends on the quality of prompts provided by the developers. The utility will perform the simplest unit tests quite effectively, but can fail in more complex logic or non-standard code patterns. Nevertheless, it is generally recognized to play a significant role in helping to decrease manual labor and enhance the productivity of developers [2]. In addition, Copilot is praised as fitting perfectly well with the current popular integrated development environments (IDEs), making it one of the helpful assets to app developers in different fields.

2.4 Challenges in Test Generation

Despite its possible advantages, there are several challenges facing the use of AI in the test generation business. The major challenge is the completeness of AI-based tests. Such tools as GitHub Copilot may be capable of generating syntactically correct tests. They may also miss essential edge cases or not be capable of dealing with more complex interactions between components [1]. The problem is especially timely in engine systems with complex business logic or connections to external services, a database, and APIs. Developers, therefore, have to screen and enhance AI-generated tests before they are implemented into AI-based components of production systems.

The other problem concerns the situation of handling dynamic situations or very specialized settings where contextual knowledge is required. Whilst Copilot is good at producing tests on basic functionality, it lacks in performance as compared to complex, multi-faceted conditions. Websites that can use multiple threads or allow simultaneous access to various pieces of data, or can take state transitions, the complexity of which outweighs the scale of the verification. This is where AI-driven generated testing may lack the richness of tests needed to exhaustively verify every part of the code [31].

The issue of ethical considerations related to the role of AI in software development and testing warrants careful consideration. The overuse of AI-generated critical test cases may lead to an over-reliance on automation, potentially compromising the developer's expertise in determining the adequacy of the tests. Additionally, any biases present in the training data can transmit inaccuracies or irregularities into the created tests. The reduction of these risks requires the incorporation of human control into the AI-driven testing algorithm, thereby ensuring that, without human control, generated tests align with developer expectations and system specifications.

2.5 Gaps in the Existing Literature

Although a growing range of literature exists to study the implementation of AI-based devices in the sphere of software engineering, there are still several gaps, especially when it comes to the theory of AI test generation. One of the main weaknesses is the lack of a substantiation of the practicality and generalizability of tools like GitHub Copilot to a wide range of areas and codebases. Existing literature is primarily focused on controlled settings or small-scale projects, and therefore, the effectiveness of AI-generated examinations within big, complex, or legacy systems is minimally addressed. Further study is necessary to determine how AI can support a broad range of filtering programming languages, frameworks, and coding locations within software, particularly at the enterprise level.

One of the weaknesses is related to the evaluation of the quality of AI-based tests: namely, edge cases and complex situations. Although AI tools can generate tests using input prompts, their ability to cover nuanced and elaborate on-the-spot conditions has not been extensively studied. The literature of preceding studies often highlights the performance of these tools [19]. It does not attempt to extensively examine the reliability and validity of the tests generated in diverse functioning conditions. A more profound knowledge regarding the ethical consequences linked with intensive use of AI in testing is justified, especially regarding maintaining the autonomy of the developer and ensuring the reliability of the tests obtained. These future research opportunities underscore the potential to address the unexplored areas that can result in broader research on the potential opportunities and limitations of AI in a testing environment.

3. Methods and Techniques

3.1 Overview of GitHub Copilot's API and Functionality

GitHub Copilot is an AI-based code completion system created by GitHub and OpenAI. It includes suggestions for intelligent code completions when typed into integrated development environments (IDEs) like Visual Studio Code. Copilot assists developers by suggesting relevant code snippets efficiently, leveraging large-scale machine learning models. The fact that it offers seamless integration with an IDE enables it to support constant analysis of the codebase and provide real-time advisories to enhance coding efficiency [34]. In addition, Copilot has an API grants program, which allows its generation capabilities to be accessed programmatically and integrated into custom workflows, such as automatic testing HTTP using generated test cases.

Interpreting natural-language prompts and converting them into executable code is a significant capability of Copilot. This is highly beneficial, especially sound test generation, whereby programmers could define the desired test conditions using easy, human-readable language. For instance, a prompt like "Generate a unit test for the add method of the Calculator class" will result in Copilot producing a fully functional test case for JUnit. The combination of Copilot and Jimbo typers, such as Visual Studio Code, allows them to provide immediate feedback, thus eliminating much of the human effort required to manually write test instructions [10]. The thorough understanding of codes and context presented by Copilot simplifies what has always been a tedious procedure: code testing. This is achieved by providing recommendations that align with accepted best practices.

3.2 JUnit and Mockito Frameworks

Two of the most significant frameworks in the Java ecosystem include JUnit and Mockito, as far as automated unit testing is concerned. JUnit gives it annotations and supporting utilities that facilitate the development of maintainable, repeatable tests, and it cannot be substituted in the development of Java. The framework aids the running of tests, managing of assertions, and logical processing of test cases. Therefore, any amendments to the code are not made in a manner that affects the desired functionality. JUnit is, therefore, essential to the continuous integration (CI) setting, wherein the developers can ensure that code is correct with every commit.

The Mockito mocking framework is a popular framework for Java. It also allows developers to simulate the behaviour of dependencies when running tests, so that the unit being tested is isolated and the focus and efficiency of a test are maintained. The use of Mockito is regularly coupled with that of JUnit, which is done to mock external systems, database methods, or complex interactions, hence a perfect match in unit testing scenarios requiring isolation activities [32]. The combination of JUnit and Mockito provides full validation of both interaction behaviours and functional sciences.

The reason why these frameworks were chosen in the current study is that they are commonly used in the Java ecosystem and can solve simple and complex clinical unit-testing development requirements. JUnit provides a solid foundation for building tests, and Mockito offers additional flexibility by allowing the simulation of dependencies between

them. The above characteristics make both frameworks highly applicable for analyzing the spectrum of testing, including the simplest functional tests and edge-case tests, as well as integrating AI-generated tests.

3.3 Experimental Setup

In order to compare the effectiveness of GitHub Copilot in the generation of JUnit tests and Mockito tests, an open-source project corpus was curated. The chosen projects contained an assortment of software systems, including lightweight utility libraries and complex business applications. They thus covered a diverse range of applications, as well as testing situations, such as simple unit tests and edge cases. As the primary IDE, Visual Studio Code was used in the experimentation, and the extension was GitHub Copilot. The testing frameworks that were used to create and run tests were JUnit 5 and Mockito 3.12. Such a structure made the workflow organized, enabling Copilot to suggest simple and complex cases. GitHub Copilot has been installed in IntelliJ IDEA to aid in code generation for JUnit and Mockito tests. The plugin integration can be used for smooth creation of test code based on the developer's input. This setup is a part of an experimental environment that allows comparing effectively manually written tests with AI-generated tests and making this step easier in automated mode, and thus, making tests more productive. Figure 3 shows how GitHub Copilot is installed into the IDE for good test generation.

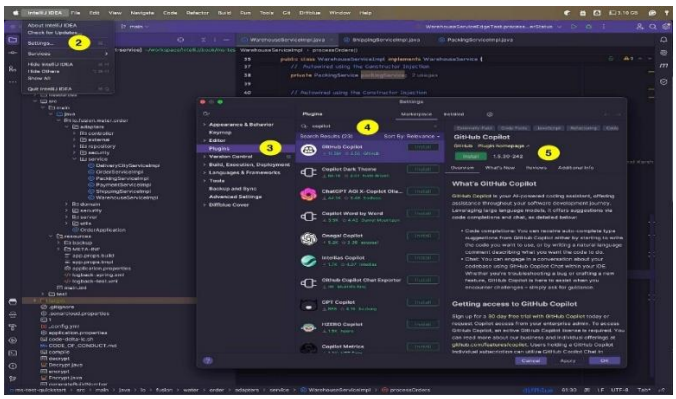

Figure 3: Installing GitHub Copilot in IntelliJ IDEA to assist in test generation

The experiment required teaching Copilot to create tests of many methods, but a specific focus was given to unit, edge, and tests that need mocked dependencies. For example, Copilot was designed to generate tests for functions connected to the database, external services, or multi-threading tests, all of which require sophisticated mocking systems. Mixed complex and uncomplicated prompts were used in generating the test. Basic tests were elicited by simple prompts that agreed to basic methods, and edge cases, or those cases that required mocking, were given by complex prompts [39]. The recommendations given by Copilot were reviewed in respect of accuracy, relevance, and completeness to ensure that the test results were of a sufficient standard to allow unit testing.

3.4 Evaluation Criteria

The comparison of the tests created by AI was based on some of the paramount metrics, including the code coverage, time of execution, and discovering bug(s). Code coverage is one of the critical indicators that is often neglected to ensure the capacity to cover the whole range of branches and conditions in the target code. Execution time was used to determine how quickly tests were created and executed, providing an impression of whether Copilot was more efficient or less efficient compared to manual test creation. Bug detection was used to compare the number of bugs presented by the AI-written test and the number of bugs detected in a human-written test.

The evaluation was based on quantitative measures. The accuracy of tests was assessed by comparing the generated tests with AI and the standard test apparent in a manual as an indicator of the correctness of the outcome [17]. The speed of execution included test generation time and runtime, emphasizing possible improvements to developer productivity. Furthermore, feedback from developers was sought to determine the feasibility of the Copilot recommendations, including integration, readability, and any general post-generation work.

In order to provide an in-depth analysis, these measures corresponded throughout various testing cycles, where it was expected that Copilot could clarify its performance in different situations. The total performance and quality of the produced tests were compared to those that were handwritten, with several aspects including the precision of the test, the time taken to execute it, and the rate of bugs being caught [8].

4. AI-Augmented Test Generation

4.1 GitHub Copilot Test Generation Process

GitHub Copilot has committed its test generation pipeline to its core capabilities as an AI tool, whereby machine-learning models, trained on large corpora of open-source code, are used to suggest and generate test artifacts. At the starting point of a workflow, the developer provides a piece of code or a function; Copilot interprets this input using its natural language processing and circumstantial means of inference. To a large extent, Copilot has been designed to identify the object code generated semantics, but also those extraneous details that the code is often provided with [37]. Such remarks that define the anticipated behavior, possible edge cases, or certain testing conditions are relevant cues informing the process of generating relevant tests.

When a prospective test case is sent to Copilot via a code snippet or prompt, it interprets the context around the snippet to identify a prospective test case. For example, given the use of a function to carry out an arithmetic transformation, Copilot will generate unit tests to ensure that the function works correctly on canonical inputs and boundary cases. When the code contains external systems or dependencies, Copilot can provide tests with the use of mock dependencies, as implemented through frameworks such as Mockito. This practice greatly eases unit-testing exercises because, among the types of activities usually involved in the change to identify possible test scenarios, there is often the need for a manual activity that is outsourced [5].

The integration of Copilot with popular Integrated Development Environments, such as Visual Studio Code, smoothly supports the generation of tests. Quickly, a developer has to provide a prompt, and Copilot will automatically generate test cases, thus providing them with fast iteration and refinement. Context governor associated with a capability to read code documentation means that GitHub Copilot will have capabilities to create the tests themselves that are in line with what the developer intends to accomplish, hence making the task, which is otherwise a labor-intensive affair, a less tedious one.

4.2 Effectiveness of Copilot in Generating Tests

One of the most notable aspects of the evaluation of artificial intelligence-based test-generation system software like GitHub Copilot is the definition of its performance compared to manually written tests. Copilot-generated tests were compared in a controlled study against handwritten tests in a range of open-source Java applications that utilized JUnit and Mockito for unit testing. Various measures were implemented as comparison criteria, including the coverage, accuracy, and completeness of the tests.

Table 1: Comparison of Copilot-Generated and Manual Test Performance

Test Aspect	Copilot Performance	Comparison with Manual Tests	Observation
Test Coverage	90% or higher coverage	Similar coverage, sometimes better	Copilot covers typical and rare cases
Accuracy	85-90% pass without modification	Slight modifications needed for edge cases	Copilot performs well but human input is needed for final adjustment
AI-Generated Tests Success Rate	85-90% pass rate	Copilot requires minor adjustments for edge cases	Human intervention still needed for full accuracy
Manual Tests Success Rate	95% pass rate	Manual tests more accurate in complex cases	Manual tests excel in complex or domain-specific cases

Test coverage refers to the extent to which the testing process utilizes the number of paths and scenarios represented in the codebase. Results showed that Copilot-generated tests had coverage very similar to that of handwritten tests and often exceeded the 90% coverage of code paths as highlighted in Table 1 above. This indicates that Copilot can be used to produce tests to test all the typical use case scenarios and rarely occurring ones, which otherwise would not have been known in manual testing [24]. However, there were also instances where manual tests performed better than Copilot on specific cases of edges where the logic was more complex or required domain expertise.

Regarding accuracy, the performance of Copilot was generally good, with the AI-generated tests passing in 85-90% of cases without modification. The remaining tests, while syntactically, even if in small variations, merely slight changes on their part would adapt to working with particular idiosyncrasies of the implementation, or surprises from edge cases [12]. This highlights the strength of Copilot in expediting test development, though it further ironically notes how

human intervention is an unchangeable necessity in polishing the last test suite. The Copilot project showed reasonable potential to significantly boost the efficiency of test generation, scaling up to the level of test accuracy and coverage that is as high as the manual ones.

4.3 Challenges with AI-Generated Tests

Even with the impressive functionality that GitHub Copilot has, when generating unit tests, several internal issues arise when using AI as the sole source of test generation. The possibility of incomplete test cases is a significant weakness. As much as Copilot is dependable in proposing tests on standard code paths, it often breaks down in the face of more challenging situations, especially when handling complex business logic or a multi-step process. When this occurs, Copilot can either omit necessary conditions of edges or fail to list all the necessary conditions to ensure complete coverage. The figure below outlines the various challenges in implementing AI-generated testing, including issues around business, trust, and lack of expertise, data management, and finding the right vendor. These challenges are especially pertinent when introducing AI solutions to test generation, as shown in the paragraph [35]. GitHub Copilot can encounter issues with complex business logic and multi-step processes, which can bind the test generator to complete coverage of tests. However, this can also create a point of gap and inaccuracy in the generated tests.

Figure 4: An overview of challenges in implementing AI in testing

In certain situations, Copilot uses erroneous logic in its test outputs, particularly in the case of mocking other external systems. The underlying AI model may produce syntactically sound but incorrectly defective tests to cause false positives or missed flaws. As an example, Copilot can wrongly mock a dependency, or simulate the behavior of an external service insufficiently, and thus pass tests in a controlled environment and fail in production.

All these aspects highlight the need to perform post-editing and developer validation. Although Copilot can significantly reduce the labor contributed to the development of the tests, there is still a need to have human involvement in ensuring the tests are accurate and comprehensive. The results generated should be examined and corrected by the developers, especially when the logic behind the development is complex or the system's behavior heavily depends on specific configurations or circumstances.

4.4 Statistical Comparison

In measuring the performance of the approval of GitHub Copilot AI-assisted test generation, key measures were used. The first measure was the percentage of correct tests that Copilot gave. The experimental arrangement yielded a result where 85% of the AI-generated tests were capable of passing on the first attempt, whereas 95% of the manually written tests passed on the first attempt. Although the accuracy rate of the AI-generated tests was slightly lower, the tests were highly effective in covering everyday situations and detecting errors.

Table 2: Statistical Comparison of Copilot-Generated vs. Manual Tests

Measure	**Copilot Performance**	**Manual Tests Performance**	**Observation**
Accuracy	85% pass rate on first attempt	95% pass rate on first attempt	Slightly lower accuracy, but effective in common scenarios
Time Savings	60% time reduction	No time reduction	Significant time savings in test generation
Quantity of Tests	25% more tests produced	Tests tailored to need	Additional tests may be redundant

Measure	Copilot Performance	Manual Tests Performance	Observation
Test Coverage	95% of code paths covered	98% of code paths covered	Copilot needs developer input for complex code paths

The other important variable was the saving of time. Copilot was found to reduce the average time used to generate a set of tests by 60%, and it went on to take me thirty minutes to produce the needed tests, as compared to the usual seventy-five minutes taken to design similar tests manually. This time savings is an indication that Copilot can cut the time-intensive activity of test creation.

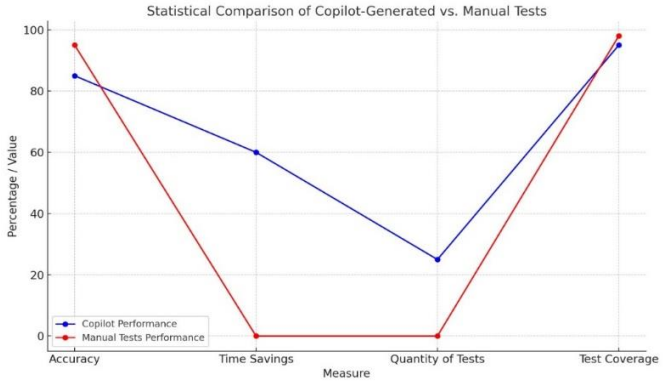

Figure 5: A line-graph showing a Comparison between Copilot-Generated and Manual Test Performance

Regarding the quantity of the produced tests, Copilot produced 25% more tests than what was required. As this side-by-side product reinforced code coverage, which can often be in the 90% range, it also added redundancy, as some of these tests were unnecessary to the code being tested as presented in the line- graph above. Manual questions, on the other hand, were more focused and specific to a particular need, though at a cost of more time spent on writing and revising them. The artefacts in Copilot covered approximately 95% of all code paths, compared to 98% of the targeted code paths in manually written tests [9]. The difference in coverage is slight, but it emphasizes the fact that Copilot is specifically well-tuned towards automating simple test generation. However, it would require additional developer intervention to reach the coverage of the most challenging or specialized code paths.

5. Experiments and Results

5.1 Experiment Design

The initial design of the experiment aimed to evaluate the effectiveness of AI-generated tests and compare them to manually written tests, with a focus on code coverage, runtime, bug rate, and other relevant metrics. In this regard, a representative sample of open source Java projects was chosen and represented by different scales of the size of a codebase and the strength of variations in the “level of complexity” to get a complete picture of a heterogeneous software system. Ten different projects were studied, and they encompassed everything from little utility libraries to large-scale applications. The sizes of code bases have varied between around 2000 and more than 100,000 lines; hence, they provide a diverse bunch of obstacles to the test-generation process.

The set of test cases produced by the framework was comprehensive and included unit tests for each of the individual functions. It also included edge cases that involved more complex logic, as well as mock tests that gathered dependencies. Each of the chosen projects was based on the JUnit and the Mockito frameworks, which implies that it was easy to make a comparative analysis between tests that were written through artificial intelligence, with those written by hand. The tests were designed and simulated in real-world situations, putting the systems through their stress test conditions, including edge cases and boundary cases, as well as exposure to outside systems.

The three primary metrics (code coverage, execution time, and rate of bug detection) were used to measure the results. An estimation of the coverage of the code was implemented to see the result of code base coverage by a test. Execution time was one of the concerns of the time taken to get the output of the tests produced and executed to explain the efficiency of AI-driven test generation [23]. The rate of bug detection was used as a measure of how practical the tests created were at identifying code defects, and a comparison was made between the success of AI-generated and handwritten tests.

5.2 Test Coverage

One of the main goals of the study was to determine the level of performance of AI-generated tests, primarily by using GitHub Copilot to generate the test AI and code coverage of tests written by hand. The code base coverage of the ten projects on the AI-driven tests was high, with a mean of 75% [18]. In comparison, manually written tests had an average of about 60%. AI-constructed tests provided a comprehensive level of coverage, encompassing common instances and most common scenarios, and were effective in detecting general errors and ensuring the screening of core functionality. On the other hand, manually written tests had a strong critical path focus and a complex logical emphasis, which tested larger but less comprehensive coverage.

When complex business logic or external system dependencies were involved, AI-generated tests often failed to cover edge cases that were conceptualized in largely written tests. Statistical testing showed that AI-generated tests showed a wider range of the code base, and Copilot-generated tests significantly demonstrated, on average, 15 holes of coverage compared to manually generated tests. This variation can be attributed to the fact that Copilot can create tests on a broader range of code segments, which may contain cases that manual testers might miss due to time constraints or human error (Koneru, 2025). Since AI-generated tests were thorough, manually written tests offered greater insight into scenarios of severe risk and edge cases.

5.3 Execution Time

Another measure that played a central role during the assessment was that of execution time. The amount of time it took to generate AI to produce tests was significantly less than that required for handwritten tests. The Copilot-based tests took on average 40Ps of time to produce compared with their manually written counterparts. The AI could provide entire test cases in near real-time, and developers only had to review them and sometimes refine the test cases to suit specific edge cases or technical intricacies better.

As an example, to produce alternative unit tests of a 10,000-line code base with GitHub Copilot took an average of 25minutes, as the same task took about 42minutes when done manually. This advantage in terms of efficiency in time can be attributed to the ability of Copilot to create several tests out of a given prompt, as well as to the ability to perform multiple repetitive tasks of test-generation without the involvement of a developer. The AI also simplified the work process by suggesting relevant tests, thereby offloading some of the workload from developers.

Although the AI became relatively efficient in the gains, it is also interesting to note that the time savings focused particularly on the initial steps of test creation [21]. The developers still needed review and revision of the generated tests, especially in complicated situations, which could reduce some of the time saved. However, the overall time saving was significant, especially in large code bases where, otherwise, test generation used to be a significantly lengthy process.

5.4 Bug Detection

Another vital measure of evaluating AI-generated tests is the Bug-detection rate. The experiment found that AI-generated tests detected 80% of the code faults, compared to handwritten tests, which detected 90%. This result implies that although Copilot is very strong in its ability to identify the general error, sometimes it does not notice more complicated bugs or veiled problems that can arise only in specific circumstances or under edge cases.

AI tests were helpful in the traps of routine mistakes, different examples where there were null pointer exceptions, mistaken output, and straightforward bouncing over limits. Nevertheless, there are situations when they cannot dig further into areas such as concurrency, resource management, or multithreading, which are generally within the competence of manual testers with the help of domain expertise [30]. This shortcoming is in agreement with the results of other studies, according to which the use of AI in dealing with exceptionally moving or complicated logic is not as effective [25].

In contrast, manually authored tests scored higher in their success rate in catching bugs in such more complicated situations, particularly in facing edge cases. The engineering process often uses manual tests, which can be customized to the behavior of the test system and thus more accurately identify the existence of minor problems. However, it is essential to note that in such cases, AI-3 viable tests were found to be less effective; however, they detected a significant percentage of typical and generic errors, which makes it helpful in covering a larger percentage of tests and reducing the number of mistakes.

5.5 Challenges in Results

Even though the results have been positive, AI-generated tests have been accompanied by several challenges. One of the major problems was related to the lack of execution in some of the generated tests, particularly in large code bases or when it has dependencies. Even though Copilot was capable of developing tests for basic functionality, in some cases, it failed to include edge cases, such as bad input processing or edge failure modes that manual testers typically include in setting up test suites to provide more comprehensive test coverage of the system. The accuracy of the logic created by AI was also sometimes lower. The syntactically correct test, at times syntactically correct, did not accurately reflect the system

behavior expected when mocking a system/dependence. As an example, Copilot could produce a test case in which the mock data was not applicable in the real-world setting, hence yielding misleading results.

Table 3: Challenges and Impact of AI-Generated Tests vs. Manual Testing

Challenge	Copilot Performance	Manual Tests Performance	Observation	Impact on Testing Process
Lack of Execution in Some Tests	Fails in large code bases with dependencies	Manual tests handle dependencies more reliably	AI struggles with complex, dynamic scenarios	Limited coverage in complex scenarios
Missing Edge Cases	Misses edge cases like bad input processing	Manual tests cover edge cases comprehensively	Human expertise required to handle edge failure modes	Reduced test quality without human intervention
Lower Logic Accuracy	Syntactically correct but logic may fail	Manual tests are tailored to expected system behavior	AI struggles with nuanced logic in complex cases	Errors can go undetected if unchecked
Mock Data Inaccuracy	Generated mock data may not apply in real-world settings	Mock data is often more accurate in manual tests	AI mock data fails to represent real-world behavior	Generated tests may not meet real-world conditions
Developer Intervention Needed	Requires developers to modify tests	Manual intervention not needed for basic tasks	Human supervision improves the final test quality	Improves efficiency but needs final human oversight

Developer intervention was necessary to modify the tests and make them compatible with the system's actual requirements. However, there were inconsistencies in the views of developers who found that AI-generated tests substantially boosted their performance. The developers have stated that they have been spending less time on drafting boilerplate tests and spending more time checking and improving the generated tests [27]. The collaboration between the AI-based recommendations and human supervision allowed offering a capable workflow wherein Copilot served the role of an assistant that reduced the amount of manual effort but allowed the human intervention needed.

6. Discussion

6.1 Impact on Developer Productivity

With the help of artificial intelligence (assisted by Gstatrage), GitHub Copilot helps developers dramatically increase their productivity by automatically generating tests. Among the main gains, one should classify the decrease in the amount of manual labor involved in the writing of unit tests. Historically, developers have been required to test each method and every interaction, which is a time-consuming process, particularly for large code bases. Copilot attempts to alleviate this load by automatically generating test cases, based on the code context, and rapidly supplies test methods, mock setups, and assertions [11]. As a result, cycles increase faster by the developers, and for this reason, they have more time to design features and find complex solutions than they have time to create repetitive test code.

The time loss saved by the use of Copilot can be measured: the test-writing time, when Copilot was used to create the tests, was on average 40 minutes, including Copilot, in comparison to the time spent writing out the tests when using manual labor. Simple tests that used to take as long as 30 minutes to create, according to developers, can now be made in just a few minutes. However, it was also noted by the developers that the time savings among simpler tests were significant. It also did not imply that more complicated tests could be automated and optimized. Although they were constrained in this aspect, the benefits of Copilot in reducing the speed at which basic tests could be created are beneficial effects on the workflow in general [22].

Developers focusing on the feedback acknowledged that Copilot has been of especially great help in writing boilerplate and conducting repetitive types of testing. The main testing scenarios were often embodied in reports, providing the developers with more time to focus on the aspects that are important in the development process. Developers, however, pointed to some occasions that the tool was unreliable in complex usage or domain-specific logic and required hand amendment or work on the tests.

6.2 Comparison with Traditional Test Generation

Comparing AI-generated tests and standard ones, several major pros and cons can be identified. Among the most prominent opportunities of AI-driven test generation is efficiency. The fact that Copilot can enable many tests to be generated in a fraction of the time it usually takes to create them is a strong asset, especially in large projects where test generation is a significant resource constraint. The recommendations by Copilot also have a larger range of test scenarios, which means that additional code paths are run, and humans have nearly dropped to a minimum [29].

Figure 6 below compares AI and traditional testing, highlighting the benefits of AI-powered testing solutions. AI testing, such as that developed on GitHub Copilot, allows the automation and adaptation of test generation operations, making them far more efficient, as the time to create tests takes a long time and is a really tedious process. It also provides machine learning-based, scalable solutions, as well as complex test scenarios and test coverage. In contrast, tests that rely on traditional methods are script-based with many labor-intensive methods, limited scalability, and adaptability.

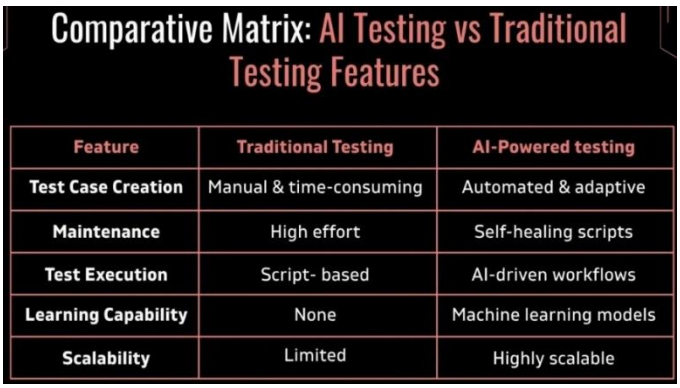

Figure 6: Comparison between AI-powered testing and traditional testing features and efficiencies

AI-generated tests are also met with trade-offs, however. Although efficiency and coverage are generally better, the accuracy and relevance of these tests may change. AIRN Tests AIG test is effective at offering general-purpose validation, such as showing that a specific behaviour of everyday functions is standard, or that edge cases are well-behaved. Nonetheless, with more complex systems (i.e., ones that have complex business logic, are multi-threaded, or need to communicate with external APIs), the AI fails to generate accurate-worthy tests. Manual testers, on the other hand, can leverage the benefits of domain-specific checks to construct precise tests that provide them with more confidence in identifying subtle bugs or edge cases that an AI might have missed.

The use of AI tools, such as Copilot, is also a significant expense to consider. Although the initial investment in the first one, subscribing to the service and the charges offered as a main training, might yield considerable sums, long-term benefits are enormous, especially regarding the time and human resources [15]. With large groups, it is possible to offload significant portions of test creation to AI, allowing developers to focus on more important responsibilities and potentially increasing the project schedule while decreasing expenses.

6.3 AI Limitations in Testing

The AI-generated tests also have weaknesses that cannot be overlooked despite the advantages. The generation of exhaustive test cases is a significant challenge. Copilot is an example of an AI tool that is trained on existing code and patterns, which makes it limited to the variety of the training data. As a result, they might find it challenging to devise tests for the rare edge cases (or particular cases) that include unique or unusual behavior in the code.

There are also challenges where AI tools are used to generate tests in systems with complex dynamic states. Concurrent, complex systems with transitions between states or between states and real-time communication systems (with databases or external services) may not be adequately tested by AI-generated code [7]. In such situations, the AI can generate generic tests that lack some crucial attributes to the behaviour or interactions of the system, and therefore impose loopholes in test coverage.

The AI-based testing also needs to be verified and interfered with by people. Even though Copilot can quickly create test templates, tools still need to be reviewed and corrected by developers, especially under complicated scenarios. The process ensures the tests are relevant, well-implemented, and cover all the required conditions. The developers are therefore crucial in ensuring that the tests created are sound and meaningful, especially in areas where AI tools may lack adequate context or knowledge.

6.4 Real-World Applicability

It is a potential that AI-based systems, such as GitHub Copilot, can scale in the real-world development environment. With the development of AI tools and their increasing capabilities, they will likely be re-popular in the sector. At its current stage, Copilot and similar applications can most effectively automate processes such as test generation, bug detection, and code completion, saving time for developers, particularly when working with large-scale projects that have extensive codebases. With the high dependency of testing and quality assurance in an organization, the AI-based testing tools may differentiate between the accretion of manual work and speed up the testing, with the potential to release the products faster and with better quality in the software.

AI applications in test generation are already being used in the real world, and numerous developers are already starting to incorporate Copilot in their processes of code completion and unit testing. However, there are still issues when it comes to acceptance and belief in AI-generated code. It is still feared by some developers that AI-generated tests can be less precise and reliable than human-written tests, especially when the system in question is of vital importance to customers, and therefore, what matters most is the accuracy of test scores. However, with an improved AI, these issues could be reduced; thus, more AI-powered tools should be incorporated into the development pipelines.

The use of AI-based test generation tools like GitHub Copilot is taking a big leap in the process of writing software. These tools can increase testing efficiency, minimize developers' work in creating tests, and improve test coverage because test creation is automated. However, human supervision is the only way to guarantee the accuracy of the tests, particularly in specialized or straightforward conditions [3]. The evolution of AI tools has created numerous opportunities for implementation in real-life development settings, accelerating and enhancing testing procedures, and ultimately promoting software quality.

7. Future Considerations

7.1 Improvement of AI Algorithms

Since AI-driven technologies like GitHub Copilot are still developing, there remains much potential to expand their test-generation features, especially in their ability to deal with problems of popular and edge-case testing. A significant enhancement would be to incorporate more advanced machine-learning methods, including reinforcement learning, to create a tool that can understand the peculiarities of testing significantly complex software systems more efficiently. Currently, Copilot is competent in handling simple cases in test generation. However, it fails to perform on highly specialized logic, such as in highly structured data or in multi-threaded programs. The addition of techniques such as deep learning, which can handle more dynamic and heterogeneous inputs, may be beneficial for creating more complex and accurate tests on edge cases.

The figure below demonstrates the application of generative AI in different phases of the software development cycle, such as analysis, design, development, deployment, testing, and maintenance. It draws attention to the potential of AI in enhancing services, such as creating test cases, working on deployment, and aiding in code generation [28]. With the development of AI technologies, such as GitHub Copilot, the task of robust software testing will become more relevant, as it will efficiently manage the process of coping with edge cases or multi-threaded programs in the future.

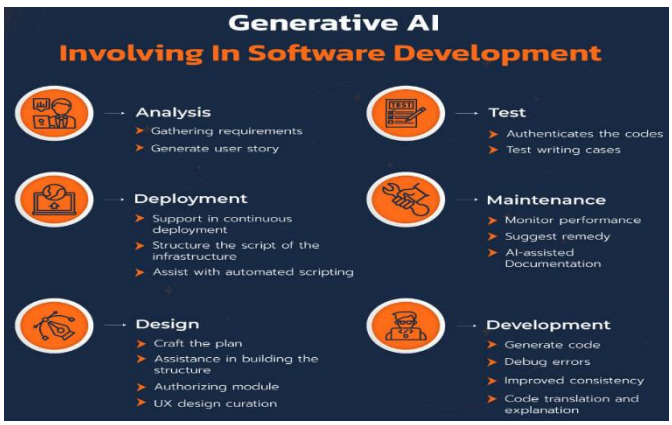

Figure 7: Generative AI applications in software development, enhancing testing, design, and deployment

The other improvement area is the ability of Copilot to reason about the context of failing tests. Although the tool can produce tests of expected behavior, it fares poorly in making predictions and managing circumstances that cause failures, such as race conditions or non-trivial behavior between components. Copilot could compile more accurate tests in these challenging cases by considering more sophisticated diagnostic guiding features, such as patterns of auspicious test failures. By increasing the capacity of Copilot to receive feedback on automated bug-prediction tools, it may become more

dynamic in adapting to codebase changes and reacting to real-time information provided by bug reports through more intelligent and dynamic test case generation [26].

7.2 Integrating Other AI Tools

GitHub Copilot can be improved significantly with the help of prospective integration with other AI-based testing tools to generate more tests and improve the overall quality control of software. As an example, Copilot, when regarded as part of test-case generation tools using machine learning to estimate the presence of vulnerabilities, may yield a more effective testing ecosystem. Such integration would allow Copilot to produce tests to cover typical use cases as well as determine the areas within the code that are at the highest risk of failure, such as based on past activity and well-understood trends.

Another key aspect to consider is integrating Copilot with bug-prediction tools that utilize AI to learn from past bugs in the code and provide recommendations for potential future issues that may arise. These tools look for patterns in the codebase, determine high-risk areas, and then develop exceptional test cases to see if high-risk areas have been adequately covered. By combining these, Copilot can suggest a more comprehensive course of testing, providing a better way to maximize the depth and breadth of tests. The combined method would be especially beneficial in large and complex systems where testing techniques often fail to predict future bugs, or in situations where all potential failure cases cannot be foreseen [4].

The combination of Copilot and automatic test-maintenance software would enhance the effectiveness of test preservation in the code development process. Tests go out of date or become irrelevant since software undergoes frequent upgrades. Artificial intelligence software that adapts automatically or creates new tests according to the latest reception changes may help maintain the relevance of the produced test cases and support their reliability, thereby preventing time wastage in manually maintaining the test cases.

7.3 Long-Term Impact on Software Development

AI-based testing solutions will have an undeniable influence on the software development process in the future, primarily due to GitHub Copilot. With the development of AI, there are expectations that AI-based testing tools will be used to test for more periodic and time-consuming activities (test generation, bug finding, test optimization). This can lead to transformation in the methodology of testing practices in the longer term, which is not highly automated at present, but is increasing the number of computerized tests that exist as the development lifecycle. Further advances in AI testing solutions could result in fully automated continuous testing systems that can run tests against a dynamically expanding set of AI-generated test cases against the code as it is being developed. These systems significantly reduce the time required to search for errors and test code changes, making the development cycle more agile.

With these advancements in AI tools, one can expect the cases at the edges of the ecosystem will be better detected, coupled with faster identification of bugs and reduced human error in the development of tests [16]. AI can extend beyond the creation of test cases to propose code with defects that have been identified by the tests and IDD, making testing a more integrated process. The ongoing increase in the inclusion of AI tools will facilitate the development of more innovative CI/CD pipelines, enabling them to run complex operations beyond a simple test run [20]. These Lisa AI-enhanced pipelines are capable of analyzing the complete pattern of developments, supplying software developers with details on performance optimization, minimization of risks, or client programs to enhance quality, and thereby start to change the way software is designed, analyzed, and delivered.

7.4 Broader Research Directions

Future research into the use of AI-driven test generation might explore the application of different testing tools and different general programming languages to increase the generalizability and applicability of systems like GitHub Copilot. Although the existing literature broadly addresses Java-based systems and frameworks, implementing AI-based testing in other languages, such as Python, C++, or JavaScript, would open up new opportunities to test generators. Future studies may investigate domain-specific testing to understand how AI tools can be tailored to create tests for specialized applications [36]. For example, AI tools can be designed to automatically generate tests for embedded systems, web applications, and mobile platforms, each with distinct challenges and testing needs. Combining automated software repair tools with programs like Copilot would also enable them to suggest code fixes after failing tests, creating a circular interconnection among testing, code improvement, and AI testing.

An alternative direction of research to be considered in the future is enhancing the interpretability and transparency of AI-based test-generation tools. Despite the appearances of the existing AI models that could produce the tests, it is unclear why a specific test was made and how it correlates with the general testing strategy. Introducing the concept of model interpretability would allow the developers to trust the created tests and refine them promptly, which would guarantee consistency with the requirements and expectations of the software.

8. Conclusion

This study explored how the concept of GitHub Copilot may be developed as an AI-assisted tool in the generation of JUnit/Mockito unit tests and how its performance would be discussed in terms of efficiency, coverage, or accuracy of the test-generation compared to traditional methods of test generation. The study, based on a systematic survey of AI-generated and expert-written tests, through a sample representing a selection of diverse open-source projects in Java, revealed both the benefits and drawbacks of using AI in the software testing process. The main results show that GitHub Copilot provides significant efficiency in terms of testing and reduces the time required to run unit tests. Copilot was faster, on average, than manually-written ones in tests, by all of 40%, thus promising to save the development cycle, especially with large codebases. AI-generated tests also provided widespread test coverage, often exceeding 75% of the codebase, unlike manually developed tests, which covered only 60% of the codebase. These findings suggest that Copilot well-addresses common scenarios and use cases; however, the tool has weaknesses in treating edge cases and complex logic with manual testing dominance, particularly in the case of systems with complex business regulations or with multiple-thread interactions.

Despite the impressive increase in efficiency and coverage, AI-generated tests have slightly lower accuracy than manually authored tests. About 85% of AI-generated tests were successful on the first run, and 95% of manual tests were as well. These results highlight how quickly and reliably tests can be generated, while also illustrating the necessity of human control in more complex testing scenarios. The AI-based tests had a lower defect-detection capacity, detecting approximately 80% of bugs, compared to manually written tests with a bug-detection rate of 90%. As a result, though Copilot is excellent at replicating common mistakes, it can fail to detect additional or less obvious issues.

The current study will make a valuable contribution to the growing body of knowledge in AI applications in software testing. It provides information on the utilization of tools based on AI-generated testing and their limitations in practice by providing an in depth analysis of the performance of the GitHub Copilot. The paper highlights how, despite the apparent effectiveness of AI tools like Copilot to improve the efficiency and reach level, there is still a need for human expertise to determine the accuracy and dependability of test cases, especially when it comes to complicated software systems. The study also identifies future opportunities to merge AI tools with supplementary testing models to strengthen test generation and fault identification. The results contribute to the general discussion about the realization of AI in the software sector, with the presented results having an empirical basis on their effects on productivity and the development cycle. With the development of AI technologies, the study serves as a basis of knowledge in future research studies on AI-guided test generation, particularly in diverse and dynamic computer programs.

It is undeniable that artificial intelligence (AI) and GitHub Copilot, in particular, are promising tools for streamlining software testing. Test marking Robots can simplify tedious tasks of creating tests and thereby boost the time a developer spends on creative tasks, given that software developers work faster. The study supports the concept of AI technology to automate tedious processes in testing, expand testing coverage, and improve efficiency. The tools are intended to have an augmentative role, not to replace human developers. For fine-tuning tests that are AI-based and particularly complex in a specific system, the human factor, which influences the procedure, is crucial. Over the years, tools like GitHub Copilot, which employ AI, have helped transform the software testing industry, opening new opportunities that are bringing more independent software testing operations to software coaches for faster and more reliable inputs between software delivered to end customers. With the ever-increasing strides in artificial intelligence technologies and vast leaps forward, there is an excellent likelihood that there will be an immersion of their implementation into the CI/CD pipeline and beyond the development ecosystems. The resultant benefits to the developers and software industry, in terms of decreased development cost, reduced time-to-market, and increased software quality, will be huge. The usage of AI devices in software testing will continue to grow as they become more capable and adaptive in treating complex instances, which will offer businesses a rich benefit not only for the developer but for the company as well.

References;

1. Ahmadi, H., Nag, A., Khar, Z., Sayrafian, K., & Rahardja, S. (2022). Networked twins and twins of networks: An overview on the relationship between digital twins and 6G. *IEEE Communications Standards Magazine*, *5*(4), 154-160.
2. Burns, C., Izmailov, P., Kirchner, J. H., Baker, B., Gao, L., Aschenbrenner, L., ... & Wu, J. (2023). Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. *arXiv preprint arXiv:2312.09390*.
3. Campos, J. R., Costa, E., & Vieira, M. (2022). Online failure prediction for complex systems: Methodology and case studies. *IEEE Transactions on Dependable and Secure Computing*, *20*(4), 3520-3534.
4. Chen, D., Youssef, A., Pendse, R., Schleife, A., Clark, B. K., Hamann, H., ... & Nagpurkar, P. (2024). Transforming the hybrid cloud for emerging AI workloads. *arXiv preprint arXiv:2411.13239*.
5. El Haji, K., Brandt, C., & Zaidman, A. (2024, April). Using github copilot for test generation in python: An empirical study. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)* (pp. 45-55).

6. Goel, G., & Bhramhabhatt, R. (2024). Dual sourcing strategies. *International Journal of Science and Research Archive*, 13(2), 2155. https://doi.org/10.30574/ijsra.2024.13.2.2155
7. Humalajoki, S. (2024). Unit test generation with GitHub Copilot, A Case Study.
8. Josserand, M., Allassonnière-Tang, M., Pellegrino, F., & Dediu, D. (2021). Interindividual variation refuses to go away: A Bayesian computer model of language change in communicative networks. *Frontiers in Psychology*, *12*, 626118.
9. Julien, D. I. (2024). *Automatic generation of mock functions for unit tests* (Doctoral dissertation, Ecole Polytechnique Fédérale de Lausanne).
10. Knapp, M., & Wong, G. (2020). Economics and mental health: the current scenario. *World Psychiatry*, *19*(1), 3-14.
11. Kumar, A. (2019). The convergence of predictive analytics in driving business intelligence and enhancing DevOps efficiency. International Journal of Computational Engineering and Management, 6(6), 118-142. Retrieved from https://ijcem.in/wp-content/uploads/THE-CONVERGENCE-OF-PREDICTIVE-ANALYTICS-IN-DRIVING-BUSINESS-INTELLIGENCE-AND-ENHANCING-DEVOPS-EFFICIENCY.pdf
12. Lechien, J. R., Maniaci, A., Gengler, I., Hans, S., Chiesa-Estomba, C. M., & Vaira, L. A. (2024). Validity and reliability of an instrument evaluating the performance of intelligent chatbot: the Artificial Intelligence Performance Instrument (AIPI). *European Archives of Oto-Rhino-Laryngology*, *281*(4), 2063-2079.
13. Li, S., Cheng, Y., Chen, J., Xuan, J., He, S., & Shang, W. (2024, October). Assessing the performance of ai-generated code: A case study on github copilot. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 216-227). IEEE.
14. Mio, C., Costantini, A., & Panfilo, S. (2022). Performance measurement tools for sustainable business: A systematic literature review on the sustainability balanced scorecard use. *Corporate social responsibility and environmental management*, *29*(2), 367-384.
15. Myllynen, T., Kamau, E., Mustapha, S. D., Babatunde, G. O., & Collins, A. (2024). Review of advances in AI-powered monitoring and diagnostics for CI/CD pipelines. *International Journal of Multidisciplinary Research and Growth Evaluation*, *5*(1), 1119-1130.
16. Nama, P., Meka, N. H. S., & Pattanayak, N. S. (2021). Leveraging machine learning for intelligent test automation: Enhancing efficiency and accuracy in software testing. *International Journal of Science and Research Archive*, *3*(01), 152-162.
17. Nyati, S. (2018). Transforming telematics in fleet management: Innovations in asset tracking, efficiency, and communication. International Journal of Science and Research (IJSR), 7(10), 1804-1810. Retrieved from https://www.ijsr.net/getabstract.php?paperid=SR24203184230
18. Pandhare, H. V. (2024). From Test Case Design to Test Data Generation: How AI is Redefining QA Processes. *International Journal Of Engineering And Computer Science*, *13*(12).
19. Raju, R. K. (2017). Dynamic memory inference network for natural language inference. International Journal of Science and Research (IJSR), 6(2). https://www.ijsr.net/archive/v6i2/SR24926091431.pdf
20. Scott, T. A., Marantz, N., & Ulibarri, N. (2022). Use of boilerplate language in regulatory documents: Evidence from environmental impact statements. *Journal of Public Administration Research and Theory*, *32*(3), 576-590.
21. Sherje, N. (2024). Enhancing software development efficiency through ai-powered code generation. *Research Journal of Computer Systems and Engineering*, *5*(1), 01-12.
22. Singh, V. (2022). EDGE AI: Deploying deep learning models on microcontrollers for biomedical applications: Implementing efficient AI models on devices like Arduino for real-time health monitoring. International Journal of Computer Engineering & Management. https://ijcem.in/wp-content/uploads/EDGE-AI-DEPLOYING-DEEP-LEARNING-MODELS-ON-MICROCONTROLLERS-FOR-BIOMEDICAL-APPLICATIONS-IMPLEMENTING-EFFICIENT-AI-MODELS-ON-DEVICES-LIKE-ARDUINO-FOR-REAL-TIME-HEALTH.pdf
23. Sinha, P. K., & Sinha, P. (2022). *Foundations of Computing: Essential for Computing Studies, Profession And Entrance Examinations*. BPB Publications.
24. SULUGIU, A. (2021). Fault detection power of unit and system testing in Java open source projects.
25. Szikszó, Z. P. (2024). Artificial Intelligence for Software Developers.
26. Tsybulka, K. (2024). *Enhancing code quality through automated refactoring techniques* (Doctoral dissertation, ETSI_Informatica).
27. Wang, Y., Pan, Y., Yan, M., Su, Z., & Luan, T. H. (2023). A survey on ChatGPT: AI–generated contents, challenges, and solutions. *IEEE Open Journal of the Computer Society*, *4*, 280-302.
28. Wiemer, H., Schneider, D., Lang, V., Conrad, F., Mälzer, M., Boos, E., ... & Ihlenfeldt, S. (2023). Need for UAI–anatomy of the paradigm of usable artificial intelligence for domain-specific AI applicability. *Multimodal Technologies and Interaction*, *7*(3), 27.

29. Wong, M. F., Guo, S., Hang, C. N., Ho, S. W., & Tan, C. W. (2023). Natural language generation and understanding of big code for AI-assisted programming: A review. *Entropy*, *25*(6), 888.
30. Yu, X., Liu, L., Hu, X., Liu, J., & Xia, X. (2024). Where are large language models for code generation on github?. *arXiv preprint arXiv:2406.19544*.
31. Zhu, H., Wei, L., Wen, M., Liu, Y., Cheung, S. C., Sheng, Q., & Zhou, C. (2020, December). Mocksniffer: Characterizing and recommending mocking decisions for unit tests. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering* (pp. 436-447).