Integration of mutation testing into unit test generation using large language models*

Andrii Kovtko[1,*,†], Volodymyr Savkiv[1,*,†], Halyna Kozbur[1,*,†], Ihor Kozbur[1,*,†] and Rostyslav Trembach[1,*,†]

[1] *Ternopil Ivan Puluj National Technical University, Ruska Str. 56, Ternopil, 46001, Ukraine*

Abstract

The article examines testing as a key component of the software development lifecycle that ensures the quality and stability of the final product. It is demonstrated that the increasing complexity of software leads to greater demands on testing as a critical phase of development. Two main approaches – manual and automated testing – are identified, with a focus placed on unit testing as the primary subject of this study. It is established that unit testing contributes to safer code changes, early defect detection, documentation, and improved code structure. At the same time, several challenges associated with writing unit tests are identified, including high time costs, maintenance difficulties, and increased load on the continuous integration and deployment system. Typical unit testing test smells are described – ineffective practices that complicate code maintenance, reduce verification accuracy, and may lead to a misleading impression of software quality. The application of artificial intelligence tools, particularly large language models (LLMs), is shown to support the automation of unit test generation, although the quality of generated tests remains inconsistent. A modified approach is proposed for integrating mutation testing into the generation of unit tests using LLMs. The concept of an automated system is presented, incorporating test generation, mutation creation, result evaluation, and iterative improvement. This integration is shown to reveal weak test coverage areas, enhance verification depth, and improve product quality. The proposed system demonstrates potential to reduce testing time, increase software stability, and offer broad applicability across various project environments.

Keywords

automated testing, unit testing, mutation testing, test smells, artificial intelligence, large language models.

1. Introduction

Software development, much like any other engineering process, comprises a series of interconnected stages that together constitute the product life cycle (Fig. 1). Each phase – from planning through to maintenance – plays a pivotal role in ensuring the quality and reliability of the final solution.

Testing represents one of the critical stages within this life cycle. It involves verifying whether the software meets specified requirements and produces the expected outcomes. Effective execution of this phase is essential for delivering a high-quality and secure product [1].

As software systems continue to grow in complexity [2], the demands placed on testing likewise intensify, establishing it as one of the central research areas in software engineering [3]. Innovation in software testing is crucial not only for enhancing product quality but also for optimizing development resources, given that testing activities may account for up to 50% of the total development budget according to various estimates [4].

* *CITI'2025: 3rd International Workshop on Computer Information Technologies in Industry 4.0, June 11–12, 2025, Ternopil, Ukraine*

[1,*] Corresponding author.

[†] These authors contributed equally.

kovtko773@gmail.com (A. Kovtko); v.b.savkiv@gmail.com (V. Savkiv); kozbur.galina@gmail.com (H. Kozbur); kozbur.igor@gmail.com (I. Kozbur); trb@tntu.edu.ua (R. Trembach)

0009-0003-0083-0514 (A. Kovtko); 0000-0001-9141-4804 (V. Savkiv); 0000-0003-3297-0776 (H. Kozbur); 0000-0002-3113-0014 (I. Kozbur); 0000-0003-4883-9393 (R. Trembach)

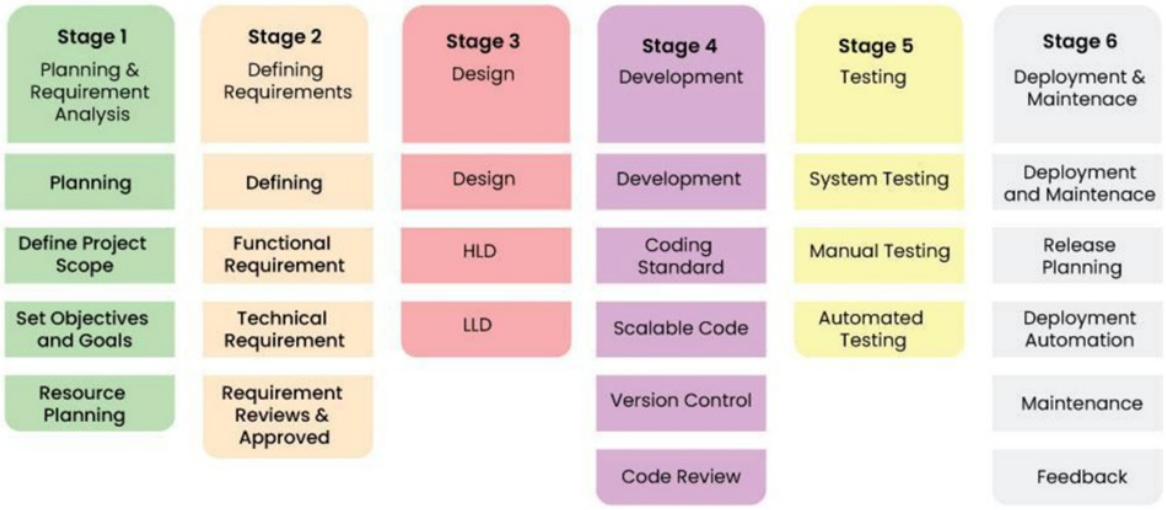

Figure 1: Software development life cycle (Source: https://media.geeksforgeeks.org/ wp-content/ uploads/20231220112830/6-Stages-of-Software-Development-Life-Cycle.jpg).

Two complementary approaches can be distinguished in software testing:

- manual testing – identifying software defects through direct interaction by a human tester [5];
- automated testing – executing tests using automation tools [6].

Manual testing remains irreplaceable in scenarios requiring flexibility, evaluation of visual components, or user interface interactions that are difficult to automate. In particular, it plays a crucial role in assessing user experience (UI/UX), where careful attention must be paid to design, navigation logic, and overall usability [7].

Automated testing is particularly effective and appropriate in cases involving repetitive, large-scale, and formalized testing scenarios [8]. It is essential for regression testing, where existing functionality must be re-verified repeatedly after code changes. Automation is also highly beneficial when testing large volumes of data or when rapid test execution across multiple environments is required – for instance, within CI/CD pipelines.

It offers significant time and resource savings, especially in long-term projects with stable requirements. Automated tests provide high verification accuracy, minimize the influence of human error, and can operate continuously without human intervention. This completes a powerful tool for maintaining software quality in large-scale or mission-critical systems.

The field of automated testing is undergoing rapid development, particularly due to the integration of artificial intelligence techniques, which enable new approaches to test generation, defect detection, and adaptive test strategy management [9], and the use of advanced high-performance computing methods for analysing results [10, 11]. Among the types of automated testing, unit testing, integration testing, system testing, and others can be distinguished (Fig. 2).

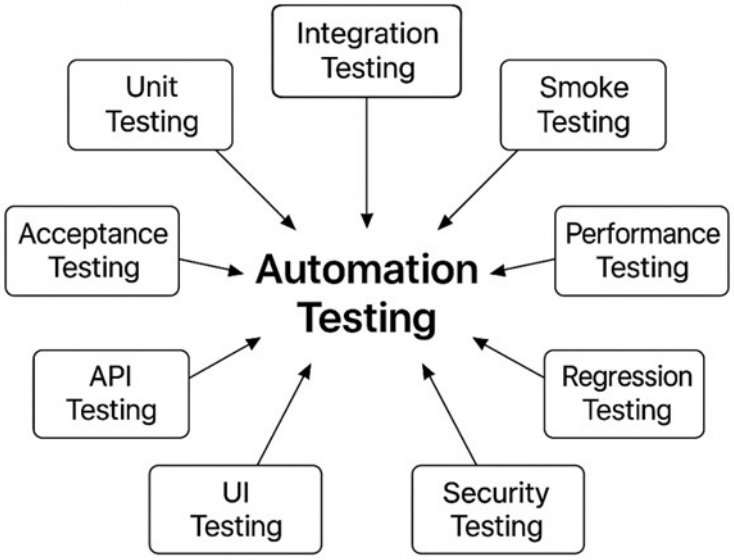

Figure 2: Types of software automated testing.

In this study, we focus specifically on automated unit testing and possible options for analysing its effectiveness using modern computing algorithms [12]. We will review existing solutions for integrating artificial intelligence into the testing process and propose our own conceptual approach aimed at improving product quality and reducing testing-related costs.

2. Unit testing

Automated unit testing involves verifying the correctness of each individual unit (function). The life cycle of unit testing is illustrated in Fig. 3. Unit tests are typically executed each time changes are made to the code.

The main advantages of unit testing include:

- improved safety and reliability of changes made to existing code;
- early detection of defects during the development process, which helps reduce the cost of fixing them;
- serving as documentation for individual units, facilitating the understanding of their interfaces and expected behavior;
- encouraging developers to write cleaner, more structured, and maintainable code.

Unit testing is an integral part of the software development process in most modern projects, demonstrating a positive impact on code quality and stability [13].

For instance, in the Linux Kernel project — one of the largest open-source initiatives, with millions of lines of code and thousands of contributors — unit testing is implemented through the kselftest framework and KernelCI system to verify the functionality of kernel components during daily builds. This enables timely detection of regressions and helps maintain system stability.

Figure 3: Unit test life cycle. (Source: https://www.lambdatest.com/dynamic-pages/resources/images/unit-testing-life-cycle.png).

In the TensorFlow project, a widely used machine learning library, unit tests cover both core computational components and the API. They are part of an automated CI pipeline that runs for each pull request, ensuring quality and consistency across different environments and versions of the library.

The study presented in [14], which analyzes over 9,000 deep learning (DL)-related projects, reveals several important patterns concerning the impact of unit testing. Specifically:

- the presence of unit tests correlates positively with key development metrics in open-source projects, such as the number of active contributors and overall developer engagement;
- codebase changes accompanied by unit tests are more likely to be accepted into the repository more quickly;
- defects in systems with adequate test coverage tend to be resolved more promptly compared to those in projects lacking unit testing.

In the study [15], which analyzed over 20,000 projects, it was found that approximately 62% of them included at least one unit test. The average number of lines of code in projects with unit tests was around 107,000, whereas projects without tests contained only about 5,605 lines. This significant difference in scale indicates that large software systems are much less likely to forgo unit testing, as the need for formalized quality control becomes critical with increasing codebase size, whereas smaller projects, which are easier to maintain manually, more often operate without formal testing practices.

Despite the numerous advantages of unit testing, including its previously discussed positive impact on code quality, project maintainability, and the onboarding of new developers, it is not a universal solution for all types of tasks. During implementation, various challenges may arise, stemming from both technical limitations and human factors. This necessitates a critical examination not only of the strengths but also of the potential weaknesses of unit testing. The key issues associated with the use of unit testing are as follows:

- Developing effective unit tests is a complex and resource-intensive task that requires a deep understanding of the system's logic and careful design of test scenarios. Thus, a study conducted by Microsoft Research found that writing unit tests typically consumes between

20% and 50% of the time allocated for developing core functionality, and in some cases up to 60% of the total development time [16].

- Test scenarios require continuous updating and maintenance, as any changes to the functionality of the program code require corresponding adjustments to the tests.
- A high level of code coverage through unit tests does not guarantee the absence of defects, as only predefined scenarios are verified, leaving some execution paths untested. A study [17] analyzing over 7,800 defects in open-source Java projects found only a weak to moderate correlation between code coverage levels and the number of defects.
- In certain cases, preparing the environment for unit testing is a technically challenging task, involving the creation of mocks to emulate external dependencies and provide access to the internal logic of units.
- The use of a large number of unit tests can significantly increase the overall testing time, which, particularly within CI/CD pipelines, may slow down the overall development and deployment cycle.

Given the aforementioned shortcomings, it is evident that the quality of unit testing is largely determined by the developer's qualifications and approach to writing tests. Poor test design and violations of recommended practices can lead to test smells — specific patterns indicative of suboptimal or problematic testing. Below are some typical situations that represent the primary causes of test smells [18]:

1. Assertion Roulette. This phenomenon occurs when a test method contains multiple assertions without proper explanations or contextual information, making it difficult to identify the cause of a potential failure.
2. Missing Assert. This situation arises when a test method does not contain any assertions, thereby stripping the test of its core verification function.
3. Empty Test. This smell appears when a test method does not contain any executable statements, making it a purely formal construct that performs no actual verification.
4. Constructor Initialization. This case takes place when a test class initializes its fields through a constructor rather than using the standard setup mechanisms provided by the testing framework.
5. Eager Test. In this case, the test verifies too many functional elements simultaneously, often invoking several production code methods within a single test, which reduces its specificity and clarity.
6. Exception Handling. This situation arises when a test uses try-catch blocks to validate exceptional behavior instead of employing specialized testing constructs designed to assert expected exceptions.
7. Conditional Test Logic. This smell occurs when assertions are placed inside conditional statements or exception handling blocks, complicating the interpretation of the test's intended behavior.

The issue of test smells has been empirically investigated in [19]. In this study, eight large open-source Java projects were analyzed. The results demonstrated a tendency for the accumulation of test smells during the evolution of software systems: for every eliminated instance of a test smell, typically two new ones emerged. The presence of such smells not only distorts the perception of test quality by artificially inflating code coverage metrics but also contributes to an increase in the number of defects within the system.

Issues related to poor test design were also addressed in [20]. This study involved a survey of 19 developers as well as an empirical analysis of 152 open-source projects. The findings highlight several important observations. First, a significant proportion of developers do not perceive poor test design as a critical problem, leading to the neglect of test code maintainability and the

effectiveness of defect detection. Second, test smells are often introduced at the early stages of test development. Finally, such shortcomings are rarely addressed during subsequent project evolution, resulting in a gradual decline in the quality of the test environment.

An overemphasis on numerical coverage metrics, combined with time constraints and variability in developers' professional skills – typical in real-world projects [21] – further escalates the emergence of numerous issues in the quality of unit testing.

3. Unit test generation

Recent changes in software testing practices are increasingly associated with the emergence of technologies that not only accelerate code verification but also redefine its very nature. Artificial intelligence and machine learning are no longer viewed merely as auxiliary tools – today, they represent an independent axis of advancement in software quality assurance [22]. These technologies now assume routine tasks, detect defects earlier than developers, and gradually reshape the perception of automation boundaries.

Among these innovations, generative models – particularly large language models (LLMs) – are demonstrating notable momentum (Fig. 4). According to Google Trends, since early 2023 there has been a significant rise in interest in LLMs and generative AI. These models are being integrated into developers' daily toolchains and are influencing how code is written, reviewed, and tested [23]. Products like GitHub Copilot not only generate code snippets — they also begin to model behavior, anticipate needs, and propose actions that would otherwise be difficult to implement manually.

Figure 4: Interest dynamics in the search terms “generative AI” and “LLM” since 2020.

In parallel with the growing adoption of LLM-based solutions in practical software testing, there has also been an intensification of scientific research efforts in this direction. For example, the study [24] proposed the tool TestPilot, which employs an adaptive approach to unit test generation involving large language models. The central idea is to construct a query for the model that includes an extended context of the target function: its name, list of parameters, comments, and examples of usage from documentation or code snippets. In cases where the generated test fails, the system formulates a refined query, incorporating the failed test itself and the corresponding error message. This allows the model to respond more precisely to the previous failure and produce an improved version of the test.

Experimental results demonstrate that this strategy leads to the generation of more meaningful, non-trivial tests that cover realistic usage scenarios. The use of three models – gpt-3.5-turbo, code-cushman-002, and StarCoder – demonstrated the superiority of the LLM-based approach over traditional methods, particularly outperforming Nessie [25] – the first unit test generation system utilizing a feedback-driven mechanism – across several key metrics, including verification completeness and test relevance.

Another example of the effective use of large language models for automated unit test generation is presented in [26], which describes a tool called ChatUniTest – a solution for automated unit test generation using ChatGPT, implemented according to the Generation-Validation-Repair approach. Its structure involves three sequential stages of test formation.

At the preprocessing stage, the system collects the most comprehensive context regarding the target code. The query includes not only the function signature and its parameters but also

associated comments, usage examples, and other available relevant fragments. The user is able to manually supplement or refine the query. During the second stage – generation – the constructed query is sent to an LLM, which produces a test or a test class based on the specified task. The final stage – post-processing – involves checking the syntactic correctness of the generated code and executing it to confirm its validity. This approach combines the flexibility of user customization with the adaptive nature of LLMs and ensures quality control at each stage of test generation.

The analysis of the described examples of automated test generation systems highlights two key challenges inherent to such approaches. First, crafting an effective prompt for the LLM is critically important, as the quality of the generated test largely depends on the amount and relevance of the provided context. Both discussed frameworks demonstrate the authors' emphasis on supplying as much detailed information about the target function as possible. Second, an essential requirement for system effectiveness is the verification of generated tests – both in terms of syntactic correctness and actual execution validity. Without this stage, test generation loses its practical value, as it does not guarantee reliable automatic coverage.

4. Automated unit test generation system

Despite their considerable advantages, automated test generation tools have several limitations that warrant close attention. In particular, the quality of generated tests tends to be inconsistent and depends on multiple factors — such as the type of unit being tested, the project's existing test coverage, and the overall structure and cleanliness of the codebase. In practice, it is not uncommon for tests generated by AI to fail to properly verify the expected behavior of functions or classes, thereby undermining the reliability of automated testing.

One of the solutions for evaluating the quality of generated tests is mutation testing [27]. The core idea of mutation testing is to deliberately introduce changes into the source code – so-called mutations – that simulate typical software defects. The purpose of unit tests in this context is to detect these artificially introduced errors. If the mutated code does not cause any test failures, this indicates a low capability of the test suite to identify defects.

Thus, mutation testing serves as an effective technique for identifying weak or superficial tests, improving the accuracy and depth of logic verification within software units. Applying this approach at the early stages of the development lifecycle not only enables the timely detection of defects but also significantly enhances overall test coverage.

In this context, a concept is proposed for an automated system for generating and improving unit tests, which combines the capabilities of artificial intelligence and mutation analysis to achieve high-quality test suites. The architecture of the system consists of the following sequential components:

1. Test Generation. At this stage, large language models (LLMs), pre-trained on source code and examples of test scenarios, are employed to automatically generate unit tests. LLMs are capable of covering a wide range of potential scenarios, including those often overlooked in manual testing.
2. Mutation Injection. The codebase is modified by introducing controlled changes – mutations – using specialized frameworks. This enables the assessment of the generated tests' ability to detect intentionally introduced defects.
3. Analysis and Refinement. Based on the evaluation of mutation testing outcomes, the AI model updates or augments the existing tests to achieve higher effectiveness.
4. Iterative Learning. A feedback loop is integrated into the system, allowing the AI to incrementally enhance the quality of the tests by learning from the outcomes of previous iterations.
5. Termination Criterion. The optimization process concludes when the differences in mutation metrics become negligible and further iterations yield no significant improvements. At this point, quantitative thresholds are defined to formalize completion.

The proposed integration of generative AI tools with mutation testing methods anticipates several key outcomes:

- Enhanced Testing Effectiveness. Combining the capabilities of artificial intelligence with the thoroughness of mutation analysis enables the formation of a test suite that more adequately covers edge cases and hidden defects. This approach facilitates deeper verification of software module behavior, even under complex or unpredictable conditions.
- Optimization of Time Expenditure. Automating the test generation process combined with an iterative refinement mechanism significantly reduces the amount of manual effort required for quality validation. This allows developers to focus on core development tasks while minimizing the time spent on analyzing and revising tests.
- Improvement of Final Product Quality. Applying the automated system at early stages of the software development lifecycle fosters the early detection of defects, reducing the cost associated with fixing issues identified at later stages and increasing the overall stability and reliability of the product.
- Versatility in Application. The architecture of the automated test generation system is designed to support multiple programming languages and testing frameworks, making it adaptable for deployment across a wide range of projects – from small libraries to large-scale distributed systems.

Conclusions

This study analyzed unit testing as a foundation for ensuring software quality, highlighting its key advantages and associated challenges. It was demonstrated that, despite its high effectiveness during the early stages of development, unit testing demands significant resources and is vulnerable to test smells.

It was established that modern generative AI models can automate the creation of tests; however, the quality of such tests remains variable. To address this issue, a modified approach combining AI with mutation testing was proposed. An architecture for an automated unit test generation system was introduced, incorporating stages of test generation, mutation analysis, and iterative refinement.

It is posited that the integration enables the identification of weaknesses in test coverage, enhances verification depth, and reduces testing costs, making it a promising solution for large-scale project implementation. Future empirical studies will be necessary to validate this hypothesis following the practical implementation of the described architecture.

Further research is planned to explore ways of leveraging mutation testing to provide additional context for test generation. Moreover, it is intended to assess existing LLMs to determine which model is best suited for test generation tasks, with the aim of developing a specialized model tailored to the needs of the proposed automated unit test generation system.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] Haiderzai M.D., Khattab M. How software testing impact the quality of software systems? // International Journal of Engineering Science. 2019. Vol. 1, No. 1. P. 1–9. doi: 10.33545/26633582.2019.v1.i1a.14.

[2] Nguyen-Duc A. The Impact of Software Complexity on Cost and Quality: A Comparative Analysis Between Open Source and Proprietary Software // International Journal on Software Engineering and Applications. 2017. Vol. 8, No. 2. P. 17–31. doi: 10.48550/arXiv.1712.00675.

[3] Salahirad A., Gay G., Mohammadi E. Mapping the structure and evolution of software testing research over the past three decades // Journal of Systems and Software. 2023. Vol. 195. Article No. 111518. doi: 10.1016/j.jss.2022.111518.

[4] Laporte C.Y., Berrhouma N., Doucet M., Palza-Vargas E. Measuring the cost of software quality of a large software project at Bombardier Transportation // Software Quality Professional. 2012. Vol. 14, No. 3. P. 4–16. URL: http://profs.etsmtl.ca/claporte/Publications/Publications/Project-at-bombardier-transportation_SQP_June%202012.pdf.

[5] Itkonen J., Mäntylä M.V., Lassenius C. How do testers do it? An exploratory study on manual testing practices // Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM 2009). 2009. P. 494–497. doi: 10.1109/ESEM.2009.5314240.

[6] Mahajan P., Shedge H., Patkar U. Automation testing in software organization // International Journal of Computer Applications Technology and Research. 2016.Vol. 5, No. 4. P. 198–201. URL: https://doi.org/10.7753/IJCATR0504.1004.

[7] Thant K. S., Tin H. H. K. The impact of manual and automatic testing on software testing efficiency and effectiveness // Indian Journal of Scientific Research. 2023. Vol. 3, No. 3. P. 88–93. URL: http://www.ijsronline.org/issue/20230714-032703.942.pdf.

[8] Kumar D., Mishra K.K. The impacts of test automation on software's cost, quality and time to market // Procedia Computer Science. 2016. Vol. 79. P. 8–15. doi: 10.1016/j.procs.2016.03.003.

[9] Alshahwan N., Chheda J., Finogenova A., Gokkaya B., Harman M., Harper I., Marginean A., Sengupta S., Wang E. Automated unit test improvement using large language models at Meta // Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE '24). 2024. P. 185–196. doi: 10.1145/3663529.3663839.

[10] Lebovka N., Cieśla M., Petryk M.,Vygornitskii N. Cooperative sequential adsorption of monomers on a square lattice in the presence of repulsive interactions between near neighbors. Physical Review E. 110, 064801 (2024). doi: 10.1103/PhysRevE.110.064801.

[11] Lebovka N.I., Bulavin L.A., Kovalchuk V.I., Petryk M.R., Vygornitskii N.V. Impact of ageing on structure of random sequential adsorption packings of discorectangles. Journal of Physics A: Mathematical and Theoretical, Vol. 57 (33), 335001(2024). doi: 10.1088/1751-8121/ad6652.

[12] Lebovka N., Petyk M., Vorobiev E. Monte Carlo simulation of dead-end diafiltration of bidispersed particle suspensions. Physical Review E. Vol.106. 064610 (2022). doi: 10.1103/PhysRevE.106.064610.

[13] Járosiová P., Chodarev S. The Impact of Unit Test Quality on Software Development: Controlled Experiment // Proceedings of the 17th International Scientific Conference on Informatics (Informatics), Poprad, Slovakia, 2024. IEEE, 2024. P. 94–99. doi:10.17605/OSF.IO/862YT.

[14] Wang H., Yu S., Chen C., Turhan B., Zhu X. Beyond accuracy: an empirical study on unit testing in open-source deep learning projects // ACM Transactions on Software Engineering and Methodology. 2024. Vol. 33, No. 4. Article 104 doi: 10.1145/3638245.

[15] Kochhar P.S., Bissyandé T.F., Lo D., Jiang L. An empirical study of adoption of software testing in open-source projects // Proceedings of the 13th International Conference on Quality Software (QSIC 2013). 2013. P. 103–112. doi: 10.1109/QSIC.2013.57.

[16] Williams L., Kudrjavets G., Nagappan N. On the effectiveness of unit test automation at Microsoft // Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE 2009). 2009. P. 81–89. doi: 10.1109/ISSRE.2009.32

[17] Rahman H., Ameen S. How is testing related to single statement bugs? // arXiv preprint arXiv:2403.18226. 2024. URL: https://arxiv.org/abs/2403.18226.

[18] Panichella A., Panichella S., Fraser G., Zaidman A., van Deursen A., Di Penta M. Test smells 20 years later: detectability, validity, and reliability // Empirical Software Engineering. 2022. Vol. 27, Article 170. doi: 10.1007/s10664-022-10207-5.

[19] Kim D.J. An empirical study on the evolution of test smell // Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020). 2020. P. 3. URL: https://djaekim.github.io/djae.io/img/EvolutionOfTestSmell.pdf.

[20] Tufano M., Palomba F., Bavota G., Penta M.D., Oliveto R., Poshyvanyk D. An empirical investigation into the nature of test smells // Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016), Singapore, 2016. P. 4–15. URL: https://www.cs.wm.edu/~mtufano/publications/C4.pdf. doi: 10.1145/2970276.2970340.

[21] Tufano M., Bavota G., Poshyvanyk D., Di Penta M., Oliveto R., De Lucia A. An empirical study on developer related factors characterizing fix-inducing commits // Journal of Software: Evolution and Process. 2017. Vol. 29, No. 1. URL: doi: 10.1002/smr.1797.

[22] Coutinho M., Marques L., Santos A., Dahia M., França C., de Souza Santos R. The role of generative AI in software development productivity: a pilot case study // Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware 2024). Porto de Galinhas, Brazil. 2024. ACM, 2024. P. 1–8. URL: https://arxiv.org/pdf/2406.00560v1. doi: 10.48550/arXiv.2406.00560.

[23] Alenezi M., Akour M. AI-driven innovations in software engineering: a review of current practices and future directions // Applied Sciences. 2025. Vol. 15, No. 3. Article 1344. URL: https://www.mdpi.com/2076-3417/15/3/1344/pdf?version=1738038423 doi:10.3390/app15031344.

[24] Schäfer M., Nadi S., Eghbali A., Tip F. An empirical evaluation of using large language models for automated unit test generation // IEEE Transactions on Software Engineering. 2024. Vol. 50, No. 1. P. 85–105. URL: https://arxiv.org/pdf/2302.06527. doi: 10.48550/arXiv.2302.06527

[25] Arteca E., Harner S., Pradel M., Tip F. Nessie: automatically testing JavaScript APIs with asynchronous callbacks // Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE 2022). Pittsburgh, PA, USA, 2022. P. 1494–1505. URL: https://dl.acm.org/doi/pdf/10.1145/3510003.3510106. doi: 10.1145/3510003.3510106

[26] Chen Y., Hu Z., Zhi C., Han J., Deng S., Yin J. ChatUniTest: a framework for LLM-based test generation // Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24). 2024. P. 572–576. URL: https://arxiv.org/pdf/2305.04764. doi: 10.48550/arXiv.2305.04764.

[27] Jia Y., Harman M. An analysis and survey of the development of mutation testing // IEEE Transactions on Software Engineering. 2011. Vol. 37, No. 5. P. 649–678. doi:10.1109/TSE.2010.62.