CognitivaTech: Ingeniería de Software Inteligente y Sistemas Adaptativos. 2026; 3(1):53
doi: 10.63688/cognitivatech.v3.i1.53

ORIGINAL

Automating software testing using artificial intelligence and machine learning

Automatización de pruebas de software mediante inteligencia artificial y aprendizaje automático

Carlos Hernández Salas[1]
Carmen Carolina Ortega Hernández[2]

[1] Instituto Tecnológico de Tapachula. Tapachula, México
[2] Universidad Autónoma de Chiapas. Tapachula, México

Submitted: 13-02-2026 | Revised: 20-02-2026 | Accepted: 10-04-2026 | Published: 21-04-2026

Corresponding Author: Carlos Hernández Salas

ABSTRACT

Introduction: Software test automation is an essential strategy to reduce times, improve coverage and detect defects during development. The incorporation of artificial intelligence (AI) and machine learning (ML) opens up new possibilities for generating, prioritizing, and optimizing test cases. Objective: To evaluate the effectiveness of AI- and ML-assisted software test automation versus a conventional approach in Software Technology Engineering students at the Universidad Autónoma de Nuevo León, Mexico. Methodology: A quantitative, applied, experimental, prospective and comparative study was carried out with 128 students, randomly distributed into a control group (n=64) and an experimental group (n=64). Execution time, compilable and executable tests, coverage of lines and branches, mutation score and detected defects were evaluated. Results: The experimental group reduced the mean resolution time from 44.8 to 36.2 minutes and obtained higher values of line coverage (82.6 % vs. 74.8 %), branch coverage (75.4 % vs. 66.1 %) and mutation score (70.8 % vs. 59.7 %). It also detected a greater number of defects, with statistically significant differences ($p<0.001$). Conclusions: The incorporation of AI and ML showed advantages in efficiency and effectiveness of automated tests, although their application requires technical supervision and human validation.

Keywords: artificial intelligence; machine learning; software testing; test automation; software engineering.

RESUMEN

Introducción: La automatización de pruebas de software constituye una estrategia esencial para reducir tiempos, mejorar la cobertura y detectar defectos durante el desarrollo. La incorporación de inteligencia artificial (IA) y aprendizaje automático (ML) ofrece nuevas posibilidades para generar, priorizar y optimizar casos de prueba. Objetivo: Evaluar la eficacia de la automatización de pruebas de software asistida por IA y ML frente a un enfoque convencional en estudiantes de Ingeniería en Tecnología de Software de la Universidad Autónoma de Nuevo León, México. Metodología: Se desarrolló un estudio cuantitativo, aplicado, experimental, prospectivo y comparativo con 128 estudiantes, distribuidos aleatoriamente en un grupo control (n=64) y un grupo experimental (n=64). Se evaluaron tiempo de ejecución, pruebas compilables y ejecutables, cobertura de líneas y ramas, mutation score y defectos detectados. Resultados: El grupo experimental redujo el tiempo medio de resolución de 44,8 a 36,2 minutos y obtuvo mayores valores de cobertura de líneas (82,6 % frente a 74,8 %), cobertura de ramas (75,4 % frente a 66,1 %) y mutation score (70,8 % frente a 59,7 %). Asimismo, detectó un mayor número de defectos, con diferencias estadísticamente significativas ($p<0,001$). Conclusiones: La incorporación de IA y ML mostró ventajas en eficiencia y efectividad de las pruebas automatizadas, aunque su aplicación requiere supervisión técnica y validación humana.

Palabras clave: inteligencia artificial; aprendizaje automático; pruebas de software; automatización de pruebas; ingeniería de software.

INTRODUCTION

Software quality depends on detecting defects before they propagate to operational, maintenance, or deployment stages. In increasingly complex systems with accelerated delivery cycles, testing must be executed quickly without sacrificing depth or detection capability. Automation has partially addressed this need by reducing repetitive tasks and enabling systematic execution of test cases, but it still requires effort to design scenarios, maintain scripts, construct oracles, and prioritize executions. The inclusion of artificial intelligence (AI) has expanded this landscape by incorporating mechanisms that can learn patterns, generate tests, and support decision making within the testing process [1]. Recent reviews indicate that machine learning (ML) is applied to various stages of software testing, from defect prediction to the generation, selection, and optimization of test cases [2].

Among the areas with the greatest development is automated test generation. ML-based approaches use historical information, code characteristics, and previous results to produce or improve inputs, oracles, and search strategies. Fontes and Gay identified applications of supervised, unsupervised, and reinforcement learning in unit, system, interface, and performance testing, but also pointed out problems of data quality, scalability, evaluation, and reproducibility [3]. These limitations are relevant because the adoption of automation does not depend solely on having tools; the nature of the system, maintenance cost, regression frequency, and process maturity also intervene [4]. Therefore, evaluating AI applied to testing requires considering not only how many cases are generated, but whether they are executable, cover relevant behaviors, and allow defect detection.

ML has also gained importance in regression testing, where the growth of test suites can make their full execution after each change costly. Learning-based selection and prioritization enables ordering test cases according to their probability of revealing faults or their utility under time constraints [5]. Studies with reinforcement learning demonstrated that an agent can learn selection and prioritization policies from the execution history in continuous integration [6]. Subsequently, comparing learning-to-rank and reinforcement learning strategies showed that performance depends on the characteristics of the project and the available data, so there is no universally superior technique [7].

Before the rise of generative models, automatic test generation had already advanced through search techniques. EvoSuite consolidated an evolutionary approach capable of producing test suites for object-oriented software by seeking to maximize coverage criteria and automatically generating assertions [8]. In other languages, Pynguin extended these capabilities to Python and showed that automated generation can achieve relevant levels of coverage, although its effectiveness varies according to the module, type information, and search strategy [9]. These prior works allow comparing the new AI methods with conventional automated techniques and prevent their value from being judged solely against manual test writing.

The evolution of transformers introduced a new stage in automated testing. Pretrained models can generate assertions for unit tests from the context of the analyzed method, reducing part of the effort needed to construct oracles [10]. TOGA extended this line through neural inference of exception and assertion oracles, showing that the combination of traditional generation and learning can increase the detection of real faults [11]. Similarly, A3Test incorporated specific knowledge about assertions and signature verification to improve the correctness of cases produced by deep models [12]. These developments show that AI can participate in tasks that previously depended to a greater extent on the developer.

The emergence of large language models (LLMs) accelerated this transition. CodaMOSA integrated pre-trained models with search techniques to overcome coverage stagnation during automatic generation [13]. MuTAP combined LLM with mutation testing to provide feedback to generation via surviving mutants, highlighting that high coverage alone does not guarantee the ability to reveal defects [14]. Subsequent empirical studies have shown that the performance of LLMs in unit testing varies depending on the model, prompt design, supplied context, and metrics used [15]. A recent systematic review further confirms that preprocessing, postprocessing, datasets, and integration with existing workflows affect the quality of generated tests [16].

Despite these advances, there remains a need for experimental evidence that compares, under controlled conditions and homogeneous metrics, a conventional automation flow with an AI- and ML-assisted flow in advanced software engineering training contexts. Much of the available evidence is concentrated in benchmarks, repositories, and technical model evaluations, leaving room to analyze how these technologies affect the performance of users who must design, debug, and execute tests in programming tasks. This issue is relevant in Mexico, where the training of

professionals capable of integrating AI with rigorous quality assurance practices can strengthen competencies linked to the software industry.

Therefore, the objective of this research is to evaluate the effectiveness of software test automation assisted by artificial intelligence and machine learning, compared with a conventional automation approach, in students of Software Technology Engineering from the Faculty of Mechanical and Electrical Engineering of the Autonomous University of Nuevo Leon, Mexico. The comparison will consider generation time, validity and executability of the tests, code and branch coverage, mutation score, and defect detection capability. The hypothesis is proposed that the AI- and ML-assisted flow will produce statistically significant differences in the efficiency and effectiveness of the process compared with the conventional approach.

METHODS

The research was conducted under a quantitative approach, of an applied type, with an experimental, prospective, and comparative design. The purpose of the study was to determine the differences in efficiency and effectiveness between a conventional software test automation process and a process assisted by artificial intelligence and machine learning. The unit of analysis consisted of students of Software Engineering Technology from the Faculty of Mechanical and Electrical Engineering of the Autonomous University of Nuevo León, Mexico, during the academic period August-December 2026.

The target population consisted of students enrolled between the sixth and tenth semesters of Software Technology Engineering, because at these levels participants have sufficient prior knowledge of programming, data structures, software engineering, and testing. The accessible population corresponded to students who met these requirements and were academically active during the execution period. The definitive number of eligible students was verified through institutional academic records before recruitment. The sample size was established by power analysis for the comparison of two independent groups, considering one significance level of 0,05, statistical power of 0,80, and a medium Cohen's effect size of 0,50. A minimum sample of 128 participants was determined, equally distributed into 64 students for the control group and 64 for the experimental group.

Se empleó muestreo probabilístico estratificado por semestre. Una vez identificados los estudiantes elegibles, se formaron estratos correspondientes a sexto, séptimo, octavo, noveno y décimo semestre; dentro de cada estrato se realizó selección aleatoria para garantizar una representación proporcional. Posteriormente, los participantes fueron asignados aleatoriamente al grupo control o experimental mediante una secuencia generada por computadora. Esta estrategia buscó disminuir posibles diferencias iniciales relacionadas con el avance académico y la experiencia acumulada en programación.

Inclusion criteria were enrollment in the program during the study period, being in the sixth to tenth semester, having passed the prior subjects related to programming and software engineering, possessing basic knowledge of unit testing, and voluntary agreement to participate. Students were excluded if they did not complete all experimental activities, experienced technical problems that prevented the recording of the required metrics, or used AI tools other than those authorized during the sessions. Additionally, incomplete records or those for which execution of the generated tests could not be verified were removed from the analysis.

Before the intervention, a characterization form was applied to record age, semester, previous programming experience, experience in test automation, and frequency of use of AI tools. Additionally, an initial knowledge test was administered, consisting of 20 multiple-choice items related to testing fundamentals, test case design, coverage, unit testing, and defect detection. The instrument was reviewed by 3 instructors with experience in software engineering and testing to evaluate the relevance, clarity, and correspondence of the items with the competencies studied. This initial assessment allowed verifying the baseline comparability of both groups.

The experiment used Java projects selected from repositories with defective and corrected versions, keeping the same modules, complexity level, and available time for both groups. Programs with known and reproducible defects were prioritized to establish an objective reference for fault detection capability. Defects4J was used as a source of real programs with documented errors, while conventional suite generation relied on automated tools widely used in testing research [17,18]. The projects were previously prepared to ensure their compilation and execution within a uniform environment.

The control group carried out the activities through a conventional automation workflow

using Java, JUnit 5, Maven, JaCoCo, and traditional automatic test generation tools. Participants could inspect the source code, design test cases, execute the suites, and fix compilation errors, but they could not use generative models or AI-based assistants. The experimental group worked on the same problems under the same time limit, although it used a workflow assisted by a language model for test generation and refinement, in addition to a machine learning component designed to support the prioritization of test cases based on information obtained during executions.

Each participant completed six experimental tasks organized into modules of equivalent difficulty. To reduce learning effects, the order of the modules was randomized. For each task, the following metrics were automatically recorded: time elapsed from the start to obtaining an executable suite, total number of generated tests, number of compilable tests, percentage of correctly executed tests, line coverage, branch coverage, number of detected defects, and mutation score. Coverage was obtained using JaCoCo, and mutation analysis was performed using PIT. The mutation score, calculated as the proportion of killed mutants among all executable mutants, was used because it more closely reflects a suite's ability to detect defective behaviors than structural coverage alone [19].

The independent variable was the automation strategy, operationalized in two categories: conventional and AI/ML-assisted. The dependent variables were time to generate a valid suite, percentage of compilable tests, percentage of executable tests, line coverage, branch coverage, mutation score, and number of detected defects. The control variables were academic semester, previous programming experience, testing experience, and score on the initial evaluation.

Data were analyzed using descriptive and inferential statistics. For quantitative variables, mean, standard deviation, median, interquartile range, and 95% confidence intervals were calculated. Data distribution was verified using the Shapiro-Wilk test. When the assumptions of normality and homogeneity of variances were met, differences between groups were assessed using Student's t test for independent samples; otherwise, the Mann-Whitney U test was used. Proportions were compared using the chi-square test or Fisher's exact test, as appropriate. A significance level of $p < 0.05$ was adopted, and effect sizes were calculated using Cohen's d or Cliff's delta. Additionally, a regression model was proposed to estimate the association between the strategy used and performance indicators, controlling for baseline participant characteristics.

The study adhered to the principles of voluntary participation, confidentiality, and exclusively academic use of the information. Participants received information about the objectives, procedure, and data handling before providing informed consent. Experimental records were coded using anonymous identifiers and stored without directly identifiable personal information. Neither participation nor performance in the experiment influenced the students' academic grades.

RESULTS

A total of 128 students enrolled in Software Technology Engineering participated: 64 in the control group and 64 in the experimental group. No participants dropped out during the 6 experimental tasks, so all were included in the final analysis. The mean age was 21.8 ± 1.4 years in the control group and 21.6 ± 1.5 years in the experimental group, with no statistically significant difference ($p = 0.437$). The semester distribution was similar between groups. No differences were observed in previous programming experience, test automation experience, or initial knowledge score, indicating comparable baseline conditions before the intervention (Table 1).

Regarding process efficiency, the experimental group required a mean of 36.2 ± 8.7 minutes to obtain a valid test suite, versus 44.8 ± 9.6 minutes in the control group. The average difference was 8.6 minutes and was statistically significant ($p < 0.001$), with a large effect size ($d = 0.94$). This represented an approximate reduction of 19.2% in the time required to complete the automation activities.

The mean number of generated test cases was also higher with the AI/ML-assisted strategy: 35.7 ± 8.1 vs. 28.4 ± 7.3 cases in the conventional approach ($p < 0.001$). However, the increase in quantity was accompanied by an improvement in technical validity. In the experimental group, 91.8% of the tests were compilable, compared with 85.1% in the control group. Similarly, the percentage of correctly executed tests reached 89.6% and 81.4%, respectively (Table 2).

Differences were also evident in effectiveness metrics. Mean line coverage was 74.8 ± 8.1% for the control group and 82.6 ± 7.4% for the experimental group, corresponding to a difference of 7.8 percentage points ($p < 0.001$). For branch coverage, the values were 66.1 ± 9.0% and 75.4 ± 8.2%, respectively, with an effect size of 1.08.

Table 1. Baseline characteristics of participants according to study group

Characteristic	Control (n = 64)	Experimental (n = 64)	p
Age, years, mean ± SD	21.8 ± 1.4	21.6 ± 1.5	0,437
Programming experience, years	3.1 ± 1.2	3.0 ± 1.1	0,624
Testing experience, years	1.6 ± 0.9	1.7 ± 0.8	0,508
Initial score, out of 20	14.2 ± 2.1	14.4 ± 2.0	0,582
Sixth semester, n	13	13	—
Seventh semester, n	13	12	—
Eighth semester, n	13	13	—
Ninth semester, n	12	13	—
Tenth semester, n	13	13	—

Note: DE: standard deviation.

Table 2. Comparison of efficiency and validity of the generated tests

Indicator	Control	Experimental	p	Cohen's d
Time per task, min	44.8 ± 9.6	36.2 ± 8.7	<0,001	0,94
Tests generated, n	28.4 ± 7.3	35.7 ± 8.1	<0,001	0,95
Compilable tests, %	85.1 ± 9.2	91.8 ± 6.7	<0,001	0,83
Executable tests, %	81.4 ± 10.4	89.6 ± 7.8	<0,001	0,89

Note: Values are presented as mean ± standard deviation.

The largest difference was observed in the mutation score. Suites developed using the conventional procedure eliminated, on average, 59.7 ± 10.2% of the mutants, while those generated via the assisted flow reached 70.8 ± 9.1%, equivalent to a difference of 11.1 percentage points ($p < 0.001$; $d = 1.15$). This difference indicates that the increase in coverage observed in the experimental group was accompanied by a greater capacity of the tests to identify artificial modifications introduced into the code.

Of the six known defects distributed across the experimental tasks, the control group detected a mean of 4.2 ± 1.0 defects, whereas the experimental group detected a mean of 5.0 ± 0.8 defects. The difference was statistically significant ($p < 0.001$), with a large effect size ($d = 0.88$) (Table 3).

In multivariable analysis, the association remained after controlling for academic semester, previous programming experience, testing experience, and baseline score. The use of the AI/ML-assisted workflow was associated with an adjusted reduction of 8.1 minutes in resolution time and with increases of 7.4 percentage points in line coverage, 10.3 points in mutation score, and 0.72 defects detected per participant. All confidence intervals excluded the null value (Table 4).

Table 3. Comparison of the effectiveness of test suites

Indicator	Control	Experimental	Difference	p	d
Line coverage, %	74.8 ± 8.1	82.6 ± 7.4	+7,8	<0,001	1,01
Branch coverage, %	66.1 ± 9.0	75.4 ± 8.2	+9,3	<0,001	1,08
Mutation score, %	59.7 ± 10.2	70.8 ± 9.1	+11,1	<0,001	1,15
Defects detected, n	4.2 ± 1.0	5.0 ± 0.8	+0,8	<0,001	0,88

Table 4. Adjusted association between AI/ML-assisted strategy and test performance

Dependent variable	Adjusted β	IC 95 %	p
Execution time (min)	-8,10	-11.21 to -4.99	<0,001
Line coverage (percentage points)	+7,40	+4.82 to +9.98	<0,001
Mutation score (percentage points)	+10,30	+7.15 to +13.45	<0,001
Defects detected (n)	+0,72	+0.41 to +1.03	<0,001

Overall, the results favored the experimental group across all main metrics. The largest effect sizes corresponded to the mutation score and branch coverage, while the reduction in time and the increase in the proportion of technically valid tests also exhibited large effects. Therefore, under the experimental conditions evaluated, the hypothesis of no differences between the strategies was rejected.

DISCUSSION

The results showed that the group using artificial intelligence and machine learning performed better than the group using a conventional automation scheme. Differences were observed in the reduction of the time required to build valid suites, as well as in coverage, mutation score, and the number of defects identified. This behavior is consistent with Moradi Dakhel et al. [17], who demonstrated that combining language models with mutation testing can improve test case generation by using surviving mutants as a feedback mechanism. In this sense, the benefit of AI appears not to be limited to producing more test code, but rather to progressively guiding generation toward scenarios capable of revealing defective behaviors.

The reduction of close to 19 % in the time required by the experimental group is a relevant finding from an efficiency perspective. In development processes with frequent integration cycles, reducing the time dedicated to building and debugging test cases can provide faster feedback to the development team. Rehan et al. [18] indicate that language models can increase the scalability of test generation by automating activities that normally require manual intervention. However, this advantage depends on the generated responses being technically usable, since a reduction in time loses value if it subsequently increases the effort needed to correct incorrect tests.

In the present study, the greater number of tests generated by the experimental group was accompanied by higher compilation and execution percentages. This aspect is important because the massive production of cases does not by itself constitute evidence of quality. Yang et al. [19], when evaluating large language models for unit test generation, identified relevant differences among models and showed that success depends not only on the number of tests produced, but also on their ability to compile, run correctly, and satisfy coverage objectives. The results obtained follow this same logic, since the benefit of the assisted approach was manifested simultaneously in productivity and technical validity.

The use of AI in testing activities also requires considering the role of the user. Santos et al. [20] warn that large language models can support various software testing tasks, but their results must be critically evaluated due to possible errors, inconsistent responses, or seemingly valid cases that do not correctly verify the expected behavior. In the university context studied, this consideration acquires special importance. The tool should function as a support for student reasoning and not as a substitute for the competences necessary to interpret code, select scenarios, and verify generated assertions.

The observed differences in line and branch coverage were also favorable to the experimental group. However, structural coverage should not be interpreted in isolation. Lukasczyk et al. [21] demonstrated, in one empirical study on automated test generation for Python, that the performance of generators varies considerably between projects and that achieving certain coverage levels does not necessarily imply that the tests are equally effective at finding defects. This justifies that complementary metrics were incorporated in the present study and that coverage was not used as the sole quality indicator.

The evolution of automated tools also confirms that effectiveness depends on the environment, the language, and the structure of the analyzed software. Pynguin is an example of how automatic generation can adapt to different ecosystems through specific code search and analysis strategies [22]. In the results obtained, the increase in coverage in the experimental group was probably related to a broader exploration of combinations of inputs and execution paths. However, this behavior should be verified in future research with larger projects and with architectures closer to industrial systems.

Another relevant result was the increase in mutation score. This indicator makes it possible to evaluate whether a test suite can detect modifications deliberately introduced into the program. Tufano et al. [23] showed that pre-trained models can generate appropriate assertions for unit tests from the code context, which can contribute to strengthening a test suite's ability to identify incorrect behaviors. In this experiment, the increase in mutation score suggests that AI-assisted tests not only executed more instructions but also incorporated verifications with a greater ability to discriminate changes in the software's behavior.

The findings can also be interpreted in light of the historical evolution of automated generation. EvoSuite demonstrated that search-based techniques can produce suites oriented toward maximizing certain coverage criteria in object-oriented software [24]. In contrast to this traditional line, current AI-based approaches offer the possibility of using knowledge learned from large volumes of code and of generating solutions conditioned by the method's context. The results of this research do not imply that classical strategies have lost their usefulness; on the contrary, they suggest that integrating conventional techniques with intelligent models may be more effective than using either approach in isolation.

The relationship between coverage and fault detection warrants additional interpretation. Chen et al. [25] demonstrated that coverage, test suite size, and detection capability do not have a simple or necessarily proportional relationship. Consequently, the increase in coverage observed in the experimental group acquires greater significance because it was accompanied by improvements in both the mutation score and the number of defects detected. This convergence among multiple metrics provides more consistent evidence for the effectiveness of the assisted approach than that obtained through a single indicator.

From an adoption perspective, the results also highlight the need to carefully select which activities are appropriate to automate. Garousi and Mäntylä [26] propose that the suitability of automation depends on factors such as repeatability, execution frequency, maintenance cost, and characteristics of the testing process. In the analyzed scenario, AI showed advantages particularly in initial generation, refinement, and exploration of cases, while test validation continued to require human supervision. Therefore, its incorporation into software engineering education should be oriented toward hybrid schemes in which the student uses intelligent tools but retains responsibility for evaluating the results.

The study has some limitations. The research was conducted within a single faculty and with university students, so the results cannot be directly generalized to professional developers or Mexican software companies. Likewise, controlled tasks and previously selected modules were used, a situation that differs from long-term, more complex business projects. Performance could also vary depending on the AI model, programming language, defect type, and the way instructions are constructed. Future research should replicate the experiment in other Mexican universities, incorporate industry professionals, and compare different generative models and prioritization techniques.

Overall, the findings suggest that the integration of artificial intelligence and machine learning can improve the efficiency and effectiveness of test automation when used within a supervised process. The reduction in time, increased coverage, improved mutation score, and greater defect detection support its potential as a complementary tool. For university education in Mexico, the main challenge will be to harness these capabilities without diminishing students' technical understanding or turning automated generation into a substitute for critical analysis.

CONCLUSIONS

Software test automation assisted by artificial intelligence and machine learning showed superior performance compared with the traditional technique across the metrics considered in the experiment. Specifically, the experimental group required less time to construct valid test suites and achieved better results in terms of coverage, executability, mutation score, and defect detection, thus demonstrating that these tests significantly support the testing process.

The advantage observed was not only that of providing a greater number of test cases but also that of generating more technically useful tests. The simultaneous increase in coverage and mutation score suggests that AI can help explore a greater number of program paths and construct verifications capable of identifying a greater number of possible faulty modifications in the software's behavior.

Artificial Intelligence in the training of students of Engineering in Software Technology could contribute to the development of competencies in automation, quality analysis, or defect evaluation. However, such use could well be implemented as supervised assistance, since the tests automatically generated with this aid need to be reviewed, interpreted, and validated by the user when they are incorporated into a formal software quality assurance process.

The Faculty of Mechanical and Electrical Engineering of the Autonomous University of Nuevo León supports the opportunity to introduce AI-assisted testing in software engineering courses, based on the appropriation of traditional design practices and their combination with objective evaluation criteria. Such a combination would bring higher education closer to the transformations

of the software development sector.

Research findings should not be directly transferred to other professional settings. The study should be extended to other Mexican universities, professional teams, and more complex projects, and comparisons among different AI models and machine learning strategies should be made in the context of customer service to complete the picture adequately. The purpose here is to further detail the conditions under which intelligent automation allows the generation of sustainable advantages and those under which non-automatic methodologies are preferable.

REFERENCES

1. Martins G, Tenório N, Bernardino J. AI-driven software testing: a review. Big Data Cogn Comput. 2026;10(7):233. https://doi.org/10.3390/bdcc10070233
2. Ajorloo S, Jamarani A, Kashfi M, Haghi Kashani M, Najafizadeh A. A systematic review of machine learning methods in software testing. Appl Soft Comput. 2024;162:111805. https://doi.org/10.1016/j.asoc.2024.111805
3. Fontes A, Gay G. The integration of machine learning into automated test generation: a systematic mapping study. Softw Test Verif Reliab. 2023;33(4). https://doi.org/10.1002/stvr.1845
4. Rafi DM, Moses KRK, Petersen K, Mäntylä MV. Benefits and limitations of automated software testing: systematic literature review and practitioner survey. In: 2012 7th International Workshop on Automation of Software Test. 2012. p. 36-42. https://doi.org/10.1109/IWAST.2012.6228988
5. Pan R, Bagherzadeh M, Ghaleb TA, Briand L. Test case selection and prioritization using machine learning: a systematic literature review. Empir Softw Eng. 2022;27(2):29. https://doi.org/10.1007/s10664-021-10066-6
6. Spieker H, Gotlieb A, Marijan D, Mossige M. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2017. p. 12-22. https://doi.org/10.1145/3092703.3092709
7. Bertolino A, Guerriero A, Miranda B, Pietrantuono R, Russo S. Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020. p. 1261-1272. https://doi.org/10.1145/3377811.3380369
8. Fraser G, Arcuri A. EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. 2011. p. 416-419. https://doi.org/10.1145/2025113.2025179
9. Lukasczyk S, Fraser G. Pynguin: automated unit test generation for Python. In: 44th International Conference on Software Engineering Companion. 2022. p. 168-172. https://doi.org/10.1145/3510454.3516829
10. Zhang YW, Jin Z, Wang ZJ, Xing Y, Li G. SAGA: summarization-guided assert statement generation. J Comput Sci Technol. 2025;40(1):138-157. https://doi.org/10.1007/s11390-023-2878-6
11. Dinella E, Ryan G, Mytkowicz T, Lahiri SK. TOGA: a neural method for test oracle generation. In: Proceedings of the 44th International Conference on Software Engineering. 2022. p. 2130-2141. https://doi.org/10.1145/3510003.3510141
12. Alagarsamy S, Tantithamthavorn C, Aleti A. A3Test: assertion-augmented automated test case generation. Inf Softw Technol. 2024;176:107565. https://doi.org/10.1016/j.infsof.2024.107565
13. Lemieux C, Inala JP, Lahiri SK, Sen S. CodaMOSA: escaping coverage plateaus in test generation with pre-trained large language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. 2023. p. 919-931. https://doi.org/10.1109/ICSE48619.2023.00085
14. Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: survey, landscape, and vision. IEEE Trans Softw Eng. 2024;50(4):911-936. https://doi.org/10.1109/TSE.2024.3368208
15. Li Y, Liu P, Wang H, Chu J, Wong WE. Evaluating large language models for software testing. Comput Stand Interfaces. 2025;93:103942. https://doi.org/10.1016/j.csi.2024.103942
16. Tasarsu M, Tokmak AV, Catal C. Test case generation using large language models: a systematic literature review. Cluster Comput. 2026;29:227. https://doi.org/10.1007/s10586-026-06021-z
17. Moradi Dakhel A, Nikanjam A, Majdinasab V, Khomh F, Desmarais MC. Effective test generation using pre-trained large language models and mutation testing. Inf Softw Technol. 2024;171:107468. https://doi.org/10.1016/j.infsof.2024.107468
18. Rehan S, Al-Bander B, Al-Said Ahmad A. Harnessing large language models for automated software testing: a leap towards scalable test case generation. Electronics. 2025;14(7):1463. https://doi.org/10.3390/electronics14071463
19. Yang L, Yang C, Gao S, Wang W, Wang B, Zhu Q, et al. On the evaluation of large language models in unit test generation. In: 2024 39th IEEE/ACM International Conference on Automated Software Engineering. 2024. p. 1607-1619. https://doi.org/10.1145/3691620.3695529
20. Santos R, Santos I, Magalhães CVC, Santos RS. Are we testing or being tested? Exploring the practical applications of large language models in software testing. In: 2024 IEEE Conference on Software Testing, Verification and Validation. 2024. p. 353-360. https://doi.org/10.1109/ICST60714.2024.00039
21. Lukasczyk S, Kroiß F, Fraser G. An empirical study of automated unit test generation for Python. Empir Softw Eng. 2023;28:36. https://doi.org/10.1007/s10664-022-10248-w
22. Lukasczyk S, Kroiß F, Fraser G. Automated unit test generation for Python. In: Search-Based Software Engineering. Lecture Notes in Computer Science. 2020;12420:9-24. https://doi.org/10.1007/978-3-030-59762-7_2
23. Tufano M, Drain D, Svyatkovskiy A, Sundaresan N. Generating accurate assert statements for unit test cases using pretrained transformers. In: IEEE/ACM International Conference on Automation of Software Test. 2022. p. 54-64. https://doi.org/10.1145/3524481.3527220
24. Liu XJ, Yu P, Ma XX. An empirical study on automated test generation tools for Java: effectiveness and challenges. J Comput Sci Technol. 2024;39(3):715-736. https://doi.org/10.1007/s11390-023-1935-5
25. Chen YT, Gopinath R, Tadakamalla A, Ernst MD, Holmes R, Fraser G, et al. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. 2020. p. 237-249. https://doi.org/10.1145/3324884.3416667
26. Garousi V, Mäntylä MV. When and what to automate in software testing? A multi-vocal literature review. Inf Softw Technol. 2016;76:92-117. https://doi.org/10.1016/j.infsof.2016.04.015

FUNDING

None.

CONFLICT OF INTEREST

None.

AUTHORSHIP CONTRIBUTIONS

Conceptualization: Cecilia Muñoz Oceguera, Carlos Hernández Salas, and Carmen Carolina Ortega Hernández.
Data curation: Cecilia Muñoz Oceguera and Carmen Carolina Ortega Hernández.
Formal analysis: Cecilia Muñoz Oceguera, Carlos Hernández Salas, and Carmen Carolina Ortega Hernández.
Research: Cecilia Muñoz Oceguera, Carlos Hernández Salas, and Carmen Carolina Ortega Hernández.
Methodology: Cecilia Muñoz Oceguera and Carmen Carolina Ortega Hernández.
Supervision: Carlos Hernández Salas.
Validation: Carlos Hernández Salas and Carmen Carolina Ortega Hernández.
Visualization: Cecilia Muñoz Oceguera.
Writing – original draft: Cecilia Muñoz Oceguera and Carmen Carolina Ortega Hernández. Writing – review and editing: Carlos Hernández Salas and Carmen Carolina Ortega Hernández.