https://www.sworldjournal.com/index.php/swj/article/view/swj35-01-043
DOI: 10.30888/2663-5712.2026-35-01-043

UDC 004.415.2:004.056.5

ANALYSIS OF THE EFFECTIVENESS OF SOFTWARE TESTING TECHNOLOGIES FOR INFORMATION SYSTEMS

Brynza N. O.
c.t.s., as.prof.
ORCID: 0000-0002-0229-2874
Simon Kuznets Kharkiv National University of Economics, Kharkiv, Nauky Ave. 9-A, 61165
Kharkiv National University of Radio Electronics, Kharkiv, Nauky Ave. 14, 61166
Lobanov K. S.
master's degree in department of applied mathematics
ORCID: 0009-0000-1986-304X
Kharkiv National University of Radio Electronics, Kharkiv, Nauky Ave. 14, 61166

Abstract. *The purpose of this work is to analyze the effectiveness of software testing. Modern approaches to software testing are analyzed, and their advantages, limitations, and areas of practical application are identified. To conduct a computational experiment, an experimental dataset in CSV format was created, containing approximately 15,000 records that characterize the testing parameters, specifically costs, execution time, and the number of defects found. Models were constructed to evaluate the economic efficiency of various testing types, enabling the determination of time limits for return on investment. Based on the results of the experiment, practical recommendations were formulated for choosing the optimal type of testing: manual testing is appropriate for small projects due to low costs, automated testing is appropriate for medium-sized projects with repetitive scenarios, and intelligent testing is suitable for large and dynamic systems where adaptability and high accuracy in detecting defects are critical.*

Keywords: *testing, software, modeling, automated testing, information technologies, intelligent testing, analysis, efficiency, manual testing*

Introduction.

System analysis of the subject area. Analyzing the system within the subject area, software testing is vital for ensuring reliability and adherence to established quality standards. [1, 2]. This process goes beyond simply performing tests; it also encompasses a variety of preparatory and supporting activities. These activities encompass requirements analysis, test planning, developing test scenarios, and preparing reporting documentation. Testing can be categorized into two main types: dynamic testing and static testing. Dynamic testing involves executing the software system or its individual components to evaluate their behavior. In contrast, static testing involves analyzing development artifacts–such as code, technical documentation, or specifications–without executing them.

This process extends beyond merely checking for compliance with specific criteria; it also considers the expectations of end-users and stakeholders. It assists in

evaluating quality levels, identifying shortcomings, minimizing potential risks, and preventing problems at an early stage. The software development life cycle model is an organized diagram that defines the order of processes, operations, and tasks throughout the project [3]. The software life cycle describes the sequence of steps from idea to operation, and there are different variants of such models, from linear to adaptive.

To achieve high software quality, a holistic approach is essential. This approach integrates agile methodologies, DevOps practices, diverse testing methods, auditing, and automation. Utilizing these combined strategies enhances development efficiency and improves the user experience, while effectively addressing the specific needs of each project.

Main text.

Manual interface testing can be a resource-intensive and time-consuming process. It involves numerous repetitive operations that are prone to human error, and as projects become more complex, it becomes increasingly difficult to ensure complete coverage. For large-scale sites with thousands of elements and functions, manual testing becomes an impossible task. For clarity, a comparison of manual and automated testing is provided in Table 1.

In these conditions, there is a need for specialized frameworks for test automation that optimize costs and increase accuracy. Automated tests enable quick processing of numerous scenarios, allowing them to be rerun as necessary and providing comprehensive coverage of functionality. [4]. This approach serves as a "protective barrier," providing rapid feedback, reducing technical debt accumulation, and facilitating project management.

It is advisable to automate first and foremost those elements of software that are difficult to verify or require considerable effort, in particular server processes, logging mechanisms, and database interactions. Automation is also effective for critical functions where rapid error detection is of paramount importance, such as repetitive operations like filling out forms with many fields, checking for incorrect input data and validation mechanisms, and lengthy end-to-end scenarios. Additionally, it is useful for

computational algorithms and verifying the correctness of search functions. At the same time, it is worth noting that modern approaches to automation, particularly solutions based on artificial intelligence, differ significantly from traditional automated testing tools.

Table 1 – Comparison of manual and automated testing

Aspect	Manual testing	Automated testing
Implementation	Performed manually by specialists, without tools	Uses scripts and software for automatic launch
Skill requirements	No programming is required, but analytical skills are required	Programming knowledge is required to create tests
Efficiency for repetition	Labor-intensive, prone to errors when performed repeatedly	Fast and accurate for regression tests, easy to repeat
Cost	Low initial costs, but high labor costs in the long term	High initial investment, but savings on repeated launches
Coverage	Flexible for exploratory and ad hoc testing, but limited in time	Extensive for routine tasks, but less adaptable to change
Errors	Prone to human error (fatigue, inattention)	Minimizes human error, but depends on the quality of scripts
Application	Suitable for unstable systems and early stages	Ideal for stable projects and regression

Testing metrics are divided into external (impact on other processes) and internal (effectiveness of the testing itself). They cover:

benefits of automation;

costs of creating and maintaining tests;

incident analysis; ratio of failures to actual defects;

execution time; number of tests cases;

pass/fail results; false positives or false negatives;

code coverage; code quality;

defect density;

component speed.

The initial investment in automation, which includes tools, advanced frameworks, and specialized engineering talent, is often significantly higher than the costs of manual efforts. However, the long-term return on investment (ROI) is primarily driven by the capacity to scale. In practice, using parameterized test suites enables us to exponentially accelerate test generation. However, from a systems perspective, we must account for 'noise' like mass failures triggered by single-point defects, which demand rigorous root-cause analysis. To quantify efficiency, I rely on failure density as a core KPI; mapping the failure-to-defect ratio allows for a precise comparison between projected coverage and actual system reliability. As the total number of automated test cases rises, both the duration of test runs and the number of automated tests increase together. This highlights the importance of test run duration more noticeably as the volume of automated test cases increases.

Leveraging AI in software testing involves integrating machine learning architectures, neural networks, and natural language processing (NLP) to transform traditional quality control methods. By moving away from static scripts, these complex algorithms facilitate a more adaptive and data-driven approach to system validation and defect prediction. These systems can operate in either semi-autonomous or fully autonomous modes, improving process productivity and stability. They achieve this by analyzing large volumes of data, uncovering hidden patterns, and predicting potential failures. In the current fast-paced SDLC, integrating AI isn't just about shrinking the testing window; it's about enabling high-order validation strategies that traditional frameworks simply cannot execute. This shift toward intelligent automation directly enhances product reliability and user satisfaction by moving beyond the limitations of manual or static scripting.

By using predictive analytics to analyze historical defect patterns, track changes, and monitor execution data, we can identify potential regressions well before they reach production. This systematic approach allows us to mitigate risks early on by pinpointing high-risk architectural components and strategically allocating resources for the greatest impact on system stability. Extended test coverage achieved through intelligent methods reduces the likelihood of missing important scenarios in key

functional modules, ensuring more complete quality control. The testing process becomes more adaptive and efficient, while also enhancing stability and reliability. This improvement positively impacts the overall quality of the software product, as well as the efficiency of its development and maintenance processes (Table 2).

Table 2 – Key differences between manual testing and AI-driven approaches

Aspect	Manual testing	AI-based testing
Test scenario creation	Developed manually by QA engineers based on predefined requirements and user behavior patterns	Created automatically using data from past projects, user interaction patterns, or ML models
Flexibility to change	Has a limited ability to adapt to code modifications, requires manual adjustment for new conditions	Automatically adapts to code updates, minimizing manual intervention
Testing speed	Slower due to the need for human control and sequential execution	Accelerated thanks to parallel launch and independent AI operation
Defective detection	Finds errors only within fixed scenarios, ignoring complex or unpredictable cases	Uses AI to analyze implicit patterns, predict vulnerabilities, and detect hidden issues that are overlooked by the classical approach
Resource utilization	Requires significant human involvement in test planning and execution	Streamline resource utilization through automation of routine tasks
Regression testing	Requires manual verification after each change in the program	Automatically restarts checks to assess system stability

Integrating AI within the CI/CD pipeline shifts quality assurance from a reactive process to a real-time feedback loop [5]. By embedding intelligent telemetry directly into the delivery stream, developers can instantly mitigate regressions, maintaining high-velocity output without compromising system integrity. Furthermore, AI automates high-level behavioral analysis, detecting anomalies and edge-case deviations that previously required extensive manual oversight and deep domain

expertise.

Although manual testing is time-consuming, it remains a valuable testing method for evaluating the complexity of a product, non-standard user behaviors, and various other aspects of the application's functionality. Automated testing offers two main advantages over manual testing: it operates much faster than manual methods and produces highly repeatable results, making it perfect for regression testing. Smart testing utilizes techniques based on Artificial Intelligence (AI) and Machine Learning (ML) to analyze test data more effectively than traditional methods. This approach enables greater adaptability to changes in the code of the programs being tested and allows for earlier detection of issues compared to conventional testing methods.

Experimental research. The objective of this study was to empirically evaluate the performance trade-offs between manual, automated, and intelligent testing paradigms. By employing a rigorous, phased experimental methodology, the research ensured procedural consistency and data reproducibility, providing a validated framework for the conclusions derived from the analytical results.

The first stage involved creating an experimental database. Due to commercial secrecy, which limits access to complete financial and time statistics for commercial projects, and the need to obtain a large sample for model training, a method was employed to generate representative synthetic data.

The generator parameters were calibrated based on an analysis of open data from GitHub repositories, as well as technical reports from leading IT companies, namely IBM and Google, for the period 2022–2025. The generation was performed in a Python environment using the NumPy library, which enabled the creation of a data array that statistically corresponds to real industry indicators, ensuring the full reproducibility of the experiment. The computational experiment utilized a structured dataset, where each record represents a distinct software project or test scenario. The data is presented in tabular format and contains the following attributes:

type – type of testing (manual, automated, or intelligent);

cost – total cost of testing, USD;

execution_time – total time required to perform test procedures, hours;

defect_accuracy – defect detection accuracy, %;

coverage – percentage of functionality covered by tests, %;

risk – project risk assessment on an expert scale;

project_complexity – project complexity level;

team_experience – level of experience of the testing team;

efficiency_class – target variable reflecting the effectiveness of the selected type of testing;

time_normalized – normalized execution time indicator used in machine learning models.

The collected data included details about the testing type, execution times of test scenarios, the number and severity of detected defects, as well as general project characteristics.

Before analyzing any data, it was processed using Python, along with the pandas and NumPy libraries. In this phase, rigorous data scrubbing was performed to eliminate noise, addressing missing values through targeted imputation or removal while standardizing continuous variables to ensure cross-metric comparability. Prior to analysis, Principal Component Analysis (PCA) applied to streamline the dataset's dimensionality and mitigate multicollinearity. This optimization reduced the attribute space by approximately 30%, effectively condensing the model's complexity without compromising the fidelity of the underlying data. PCA provided for increased stability of the analysis models, as well as reduced computational requirements.

The second stage of the computational experiment involved a comparative analysis of the main types of testing based on quantitative criteria. The main evaluation indicators were testing costs, defect detection accuracy, and test procedure execution time. For manual testing, data from TestRail system reports was analyzed. For automated testing, test execution logs generated by Selenium and JUnit tools were analyzed. For intelligent testing, output data from the Testim and Applitools platforms was analyzed.

To evaluate the economic viability of disparate testing paradigms, I developed mathematical models that integrate upfront capital expenditures, operational overhead,

and lifecycle payback trajectories. This analytical framework accounts for the shift from high initial investment in automation and AI toward long-term cost-efficiency and accelerated break-even points. The primary measure of effectiveness is the return on investment (ROI) ratio, defined as the ratio of saved costs to total testing costs using the following formula [6]:

$$C_{total} = C_{init} + C_{oper} * T + C_{risk} \quad (1)$$

$$ROI = \left(\frac{B - C_{total}}{C_{total}}\right) * 100, \quad (2)$$

where C_{init} – initial investment, C_{oper} – operating expenses per month, T – period (in months), C_{risk} – risk costs, B – expected reward.

Initial parameter values for each type of testing:

Manual: C_{init} = 0 USD, C_{oper} = 3200 USD/month (based on an hourly rate of \$20/hour * 160 hours), T = 6 months, C_{risk} = 1000 USD, B = 15000 USD.

Automated: C_{init} = 10000 USD, C_{oper} = 500 USD/month, T = 6 months, C_{risk} = 500 USD, B = 30000 USD.

Intelligent: C_{init} = 30000 USD, C_{oper} = 200 USD/month, T = 6 months, C_{risk} = 200 USD, B = 50000 USD.

The resulting calculation values are shown in Table 3.

Table 3 – Key differences between manual testing and AI-driven approaches

Testing type	Total cost (USD)	ROI (%)
Manual	20 200.00	-25.74
Automated	13 500.00	122.22
Intelligent	31 400.00	59.24

A negative ROI for manual testing indicates potential risks in small projects with low expected benefits, while automated and intelligent testing achieve positive values after 4–18 months, due to higher initial investments but higher efficiency. To visualize the distribution of costs, a histogram is provided (Fig. 1), which illustrates the concentration of costs for intelligent testing at around USD 30,000, with less dispersion compared to manual testing (average approximately USD 2,000).

The intelligent modeling process utilized machine learning techniques from the scikit-learn library to identify the scale and characteristics of each project, thus enabling the classification of each project as either small, medium, or large, utilizing a combination of quantifiable project attributes such as testing volume, test run time, and the total number of defects identified [7].

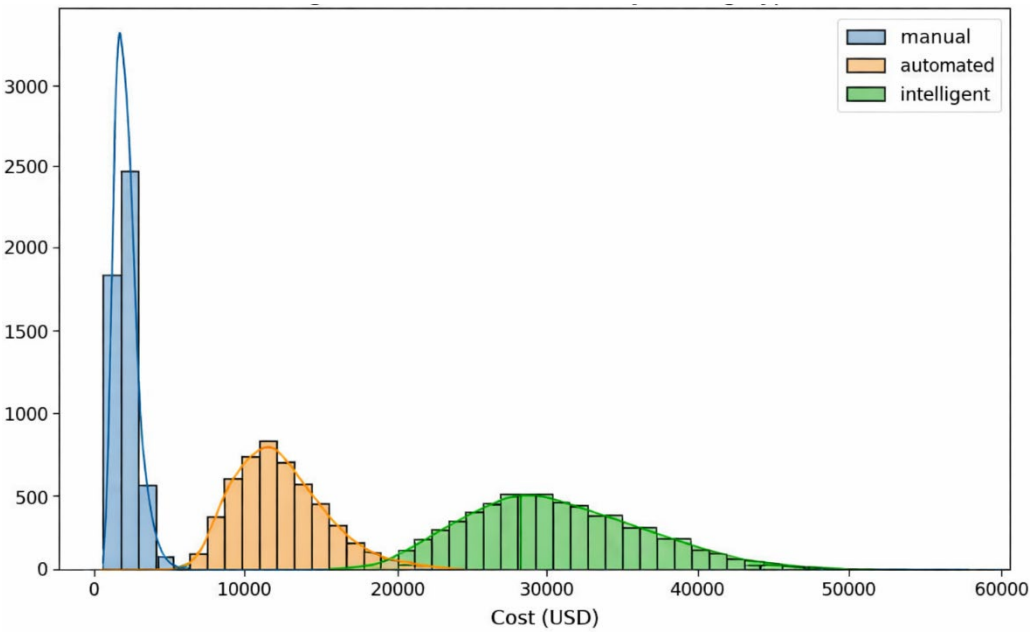

Figure 1 - Histogram of cost distribution by test type

The cluster visualization (Fig. 2) clearly distinguishes between the different project sizes, based on the scale and cost of resources required for each project.

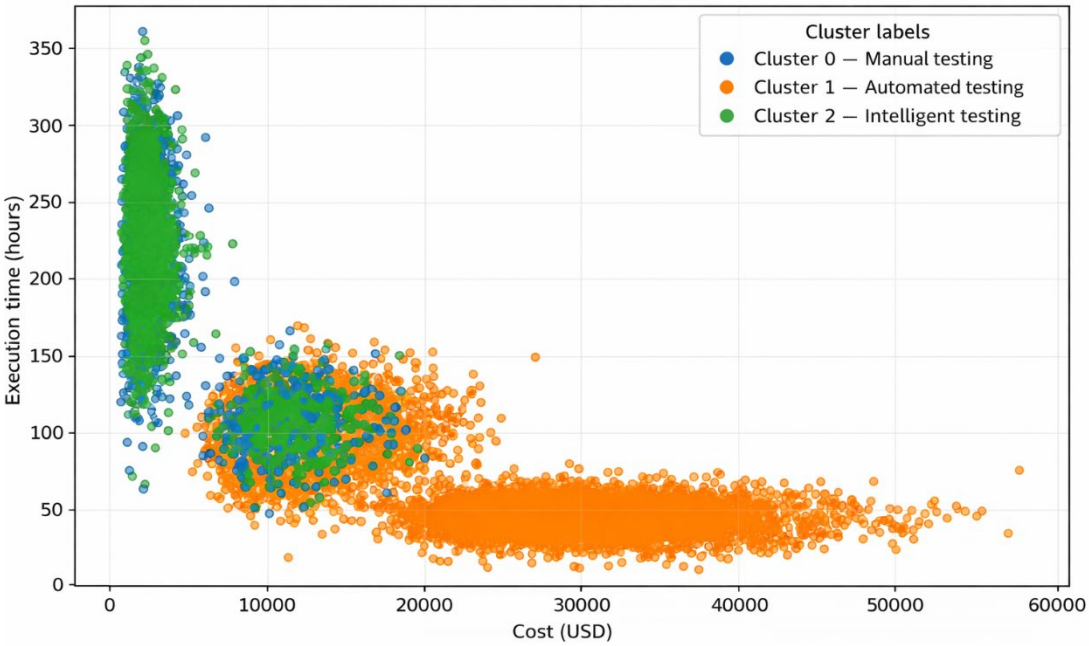

Figure 2 - Results of project clustering by testing type using the k-means method

Random Forest and Support Vector Machine (SVM) algorithms were utilized to predict testing effectiveness [8]. The models were trained in historical data, and their quality was assessed using 5-fold cross-validation. The average cross-validation accuracy was 0.839; for classification – precision (class 0)=0.82, recall=0.78, F1-score=0.80; for class 1 – precision=0.85, recall=0.88, F1-score=0.86. Validation of the derived metrics confirmed model stability, with the confusion matrix indicating a mean classification accuracy of approximately 85%. These results demonstrate that the models maintain robust predictive power across the dataset, successfully balancing precision and recall within the targeted operational parameters (Fig. 3).

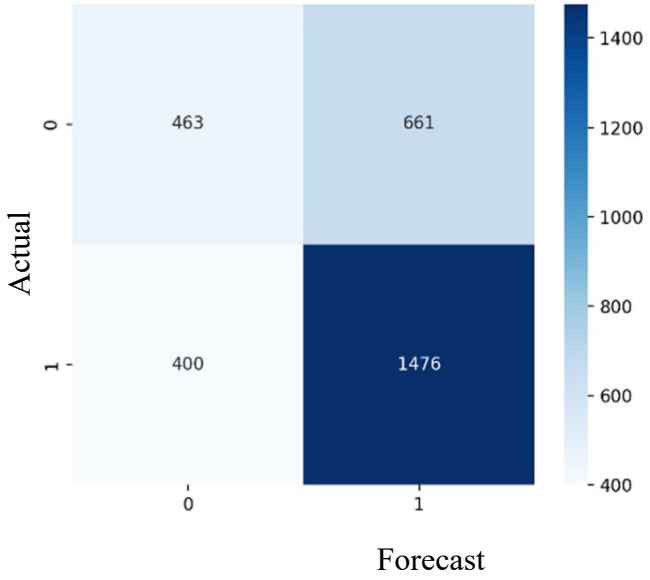

Figure 3 - Confusion matrix for the test effectiveness prediction model

To visualize the variance in performance across different testing paradigms, a boxplot of defect detection accuracy was constructed. This distribution analysis highlights the median precision of each method while clearly identifying outliers and the spread of detection consistency, providing a more granular view of reliability than simple averages (Fig. 4).

The median defect detection accuracy for manual testing is approximately 62%, indicating the limited effectiveness of this approach as project complexity increases. For automated testing, the median increases to approximately 80%, confirming its ability to provide more stable and reproducible results. The highest performance is demonstrated by intelligent testing, for which median accuracy reaches about 89%,

indicating its high effectiveness in detecting defects.

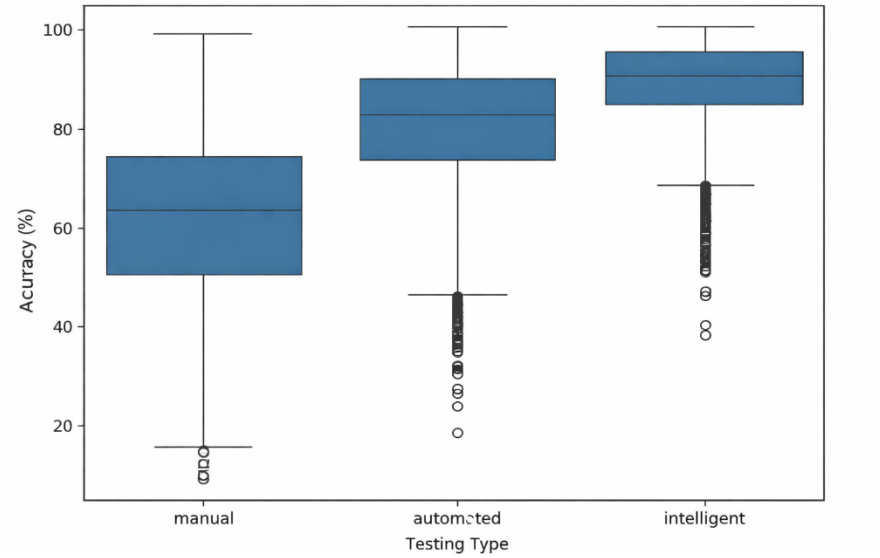

Figure 4 – Boxplot of defect detection accuracy by test type

To optimize the selection of a testing paradigm, I recommend leveraging machine learning models that weigh project scale against specific architectural characteristics. This data-driven approach allows for a dynamic assessment of whether a project's complexity warrants the high-level integration of AI or if traditional automation remains the most cost-effective path for the given scope.

Conclusions.

The simulation results support that testing efficiency is influenced by both the size and complexity of a software project. These findings allow us to suggest using manual testing for small-scale projects, automated testing for medium-sized projects, and intelligent testing for large or dynamically changing systems, which supports the applicability of these findings in practice.

Machine learning algorithm-based intelligent modeling enabled grouping projects based on their scale and complexity, with an accuracy of predicting testing effectiveness as high as 85%.

References:

1. Software Testing Process Guide: Types of Testing, Stages, and Best Practices.

URL: https://testfort.com/blog/software-qa-testing-process-overview-types-and-process-stages.

2. Chatzipetrou, E., & Varvaropoulos, K. (2024). Managerial Digitalisation Cost in the Hotel Sector: The Case of Northern Greece. Administrative Sciences, 14(3), 52.

3. Jorgensen P. C. (2014). Software Testing: A Craftsman's Approach (4th ed.). Boca Raton: Auerbach Pub. 527 p.

4. Going Deeper into the Page Object Model. URL: https://medium.com/@blakenorrish/going-deeper-into-the-page-object-model-4aee634d9c98.

5. Дуда, О., Шаклеіна, І., & Лучкевич, М. (2025). Підвищення ефективності devops за рахунок використання штучного інтелекту та машинного навчання. Herald of Khmelnytskyi National University. Technical Sciences, 351(3.1), pp. 143-149. https://doi.org/10.31891/2307-5732-2025-351-17.

6. Khankhoje R. (2023). Quantifying Success: Measuring ROI in Test Automation. Journal of Technology and Systems, 5, pp. 1–14. https://doi.org/10.47941/jts.1512.

7. Haug, Michael, Eric W. Olsen, and Luisa Consolini, eds. Software quality approaches: testing, verification, and validation: software best practice 1. Vol. 1. Springer Science & Business Media, 2001.

8. Zaidi, Hijab Zehra, et al. "Machine Learning Approaches for Software Defect Prediction." Applied Computational Intelligence and Soft Computing 2025.1 (2025): 7933078. https://doi.org/10.1155/acis/7933078.

Article sent: 14.01.2026