Partha Mohapatra

Intelligent Assurance

Artificial Intelligence-Powered Software Testing in the Modern Development Lifecycle

DeepScience

Intelligent Assurance: Artificial Intelligence-Powered Software Testing in the Modern Development Lifecycle

Partha Mohapatra
AT&T Corporation

DeepScience

Published, marketed, and distributed by:

Deep Science Publishing, 2025
USA | UK | India | Turkey
Reg. No. MH-33-0523625
www.deepscienceresearch.com
editor@deepscienceresearch.com
WhatsApp: +91 7977171947

ISBN: 978-93-7185-117-6

E-ISBN: 978-93-7185-046-9

https://doi.org/10.70593/978-93-7185-046-9

Citation: Mohapatra, P. (2025). *Intelligent Assurance: Artificial Intelligence-Powered Software Testing in the Modern Development Lifecycle*. Deep Science Publishing. https://doi.org/10.70593/978-93-7185-046-9

Preface

Traditional testing can’t match the speed and reliability demanded in modern software development and releases. This book explains the integration of Artificial Intelligence is transforming software testing—enabling smarter, faster, and more scalable quality assurance across the development lifecycle. This book demystifies the integration of AI, machine learning, and natural language processing into modern testing workflows. From automated test case generation and defect prediction to adaptive test maintenance and intelligent prioritization, it offers practical insights and real-world applications that empower QA teams to deliver higher-quality software with greater efficiency. Through a structured, hands-on approach, readers will explore AI-driven testing strategies, tools, and architectures that align with DevOps and agile practices. The book also delves into ethical considerations, challenges in AI adoption, and the future of autonomous testing agents. Whether you're a software tester, QA lead, DevOps engineer, or technology decision-maker, this book equips you with the knowledge to embrace AI-driven testing as a strategic advantage in delivering resilient, secure, and high-performance software.

Partha Mohapatra

Table of Contents

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 1: The Evolution of Software Testing: Manual Testing, Verification Methods, and SDLC Integration in Software Engineering

Partha Mohapatra

AT&T Corporation

1. Introduction to Software Testing

Testing is the formal process of verifying and validating the software product. Verification ensures that the software meets its specifications [1-3]. Validation ensures that the software built is traceable to customer requirements. Since requirements and specification documents change frequently, test cases should be revised accordingly [2,4]. Test plan is a formal document defining the testing scope and approach. It maintains communication about planned testing activities with management and enhancements. The test plan is prepared by the test leader/manager by incorporating resources required, test environment, and deliverables.

2. Historical Overview of Software Testing

Art, science or craft, one name and different meanings; testing is always a topic of discussion and themes in nearly all walks of life situations [5-8]. Testing expanded for some reason, a dream, an idea, long desired or actual behaviour, performance or functioning. Testing also means improvement of things in order to make better than it was, it is method behind new idea, new technology, new work or new inventions. Whether testing termed as an art, science or craft, analysis of a piece of art ishelpful for the painter or artist. Through tests in science, theory is proposed and in craft work improvement is done. Testing has great importance in the field of art as an analysis work, testing is base for inventing new theories or a base for changing some knowledge in science, and in crafts testing is essential for improvement.

In the present-day scenario, software testing is an integral part of software development life cycle process (SDLC) and corresponds to an art, science and craft combined. Software testing is an art since disciplined and systematic analytical skills are applied to

test a software package. It is science because theories are proposed and tested through experimentation. Finally, it is a craft in the sense that testing involves considerable skill, experience and is largely driven by intuition. So, software testing function is that testing is not only executing software but also predicts the performance of software or non-execution behavior of software itself. The predicting ability of software testing function makes software testing function different from other testing functions such as painters, scientists, crafts, etc.

3. Manual Testing Techniques

Testing is the compass that guides software development. It helps to check the problems in the software, by following some pre-defined standards [6,9]. Software testing is a process which is used for detecting the correctness, completeness and quality of developed computer software [10-12]. This process also provides an objective, independent view of the software to allow the business or users to understand the risks of software implementation. Manual testing is a very popular approach for the quality assurance of software products, deals with manual execution of test cases without using any automated tools.

Techniques of performing manual testing can be classified into different types such as Black Box Testing, White Box Testing, and Grey Box Testing [7,13-16]. Black Box Testing is related to testing the internal structure, design, and coding of the software and is also called as Structural Testing. In Black box testing maximum number of test cases should be temprary tested. White Box Testing is related to testing the working of components and their linkage, interfaces between the components. It is also called as Functional Testing. Grey box testing is originating from both White and Black box testing testing approaches. It takes advantages and tries to overcome the disadvantages of both techniques.

4. Transition to Automated Testing

Manual testing has gradually been abandoned, today all tests are put into an automated regression suite that is run overnight [2,17-19]. This approach enables a rapid development cycle and also makes it possible for a few people to maintain a large product. This technique also allows people to always know what is wrong, so the answers can be pondered during the day rather than digging when the need arises.

Although automation is a huge help, it is not a panacea. Shifting from manual to automated testing requires an investment of time — and the discipline of the automated tester can be the most difficult but important ingredient. Manual testing performed in the

blink of an eye is not the same as a well-thought-out and carefully examined automation test. The latter is certainly no less momentous but is frequently overshadowed by the former's lack of organization and likely lack of scope. In the twilight era of manual testing, the focus tends to be more on the brute-force aspect — covering as many different cases as possible — than on an articulate expression of intent when a bug is detected. While the tools are now largely in place, the mindset and discipline need to catch up.

4.1. Benefits of Automated Testing

Automated testing is the process of applying software and support resources for executing tests automatically and managing test data without manual intervention [3,20-23]. Test automation involves using automated scripts for the execution of software tests. There are important considerations for unit-level, integration, and functional tests [9,24-26].

Test automation contributes to the speed and quality of software delivery by eliminating manual operations such as the application of test data, ordering procedures, and comparing actual test results to expected results [27-29]. The benefits include improved accuracy through the elimination of human error; enhanced test coverage that incorporates areas difficult to reach with manual testing; greater scales of simultaneous execution; increased replayability enabling the same tests to be rerun numerous times through a fully automated process; and provisioning for load testing which replicates multiple users.

4.2. Tools and Frameworks for Automation

Test automation provides the promise of being able to re-run tests often that are difficult to do manually; and the hope is that that re-running of tests in an automated way will find more bugs [30-32]. Test automation adds significant cost and risk to the development process; in return for the potential of an increase in the quality of the release. Is it worth it?

Evaluating any technology investment, including test automation, incorporates business factors (budget, deadlines, resource costs, risk appetites) as well as technical factors (code and deployment complexity, the presence of APIs, product cadence). Automated testing is a program that needs to be regularly evaluated to ensure it is not hindering release velocity [9,33-35]. Allan Kelly explored the advantages and disadvantages of test automation and put forward questions to ask when evaluating test automation to determine if it is still worth it.

There are many different tools companies can use for automated testing. Some companies choose to build their own frameworks and provide training and oversight directly. Depending on your environment and your goals, the cost and maintenance of those frameworks and scripts may make SaaS products appealing. For many companies, investing in a SaaS product can provide features beyond just automation. When evaluating automated testing tools, Laura Stone provided a long list of questions that can help focus efforts and make sure that the tool chosen meets both current and future needs.

4.3. Challenges in Automated Testing

The automation of testing offers exceptional speed and has become commonplace. Still, the complex nature of software can render automation less practical [36-38]. Completely automating testing remains essentially impossible. Testing embodies a repetitive, human-driven process. Its core function is to validate that a software's state matches the intended or expected state. This is achieved by taxation in a literal sense. Executes tax of commands. Verifies the state of software. Human testing and its automation mimic intelligence: applying a set of procedures to determine specific responses. For example, making a decision, determining scores, evaluating correctness, or traversing logic paths. With software changes, test designs and their expected results must evolve accordingly. Additionally, developers must address problems that arise during test execution. Because it demands intelligence and insight, fiscal testing still requires human expertise. Yet, automation has proven invaluable for performing these tasks at exceptional speed.

5. The Rise of AI-Driven Testing

The use of AI in software testing serves a clear purpose: to enhance productivity and increase the fault detection ability of the testing activities [3,39-41]. It aims to detect a higher number of faults in less time, resulting in less time consumed for testing activities and thereby more time for development activities. In the end, the goal is to produce a high-quality product within the planned time and budget. It is important to understand that AI is only a tool and not a complete replacement for professionals in software testing. Its role is to assist the tester in the process, not to replace them.

Nowadays, companies are increasingly driven to innovate faster, necessitating the promotion of AI initiative adoption throughout the SDLC life. They are looking for an effective AI-based test approach to reduce testing time and effort by applying AI techniques. By this method, testing engineers can identify bugs early in the test development life cycle. AI techniques conserve human effort by using AI. The application of AI in software testing leads to cost reduction, time economy, and high-quality product.

5.1. Defining AI in Software Testing

Software Testing, particularly testing powered by Artificial Intelligence, is a broad area involving various concepts and ideas. The first step towards understanding the terms "Testing powered by AI" is to define the keywords separately and then combine them.AI is an umbrella term for technologies that enable machines to behave in ways that humans would describe as intelligent. AI is a resurgent field of computer science which is concerned with automation of intelligent behavior.Computer Systems which use their Programming and Algorithms, advance their knowledge and become capable of taking actions as a Human would do are referred to as Systems powered by Artificial Intelligence.

Testing powered by Artificial Intelligence is a Software Testing Procedure which is performed on an Intelligent System [36,42-44]. More precisely, it is the set of Software Testing methods that are performed to test an Intelligent System or the Intelligent methods which help in the process of Software Testing. It is not a new Testing Technique but a collection of Methods that use Intelligence during the Testing Procedure. Key attributes of Intelligence that Testing can use or that Intelligent systems posses are common sense, knowledge, problem solving, planning, learning, communication, reasoning, and the ability to move and manipulate objects.

5.2. Benefits of AI-Driven Testing

Tests are often acknowledged to be important, but also time-consuming and frustrating, not adding to the perceived value of the product. AI tools help testers avoid these tedious tasks and focus on creative testing [6,9]. AI is also increasingly being used in the broader field of software engineering, for example to generate code, fix bugs, or analyze requirements specification documents. In testing, AI-driven automation offers a range of beneficial features including self-maintenance, detailed reports, and self-analysis.

Self-maintenance refers to the test case autonomy achieved through AI. Test cases are capable of analyzing the application under test and automatically updating the test case to match changes. This allows for continuous testing in DevOps pipelines without the need for test code maintenance. Self-updating can occur at both the user interface and API levels. The testing tool can analyze the development changes and identify potential error scenarios, additionally completing areas of the test case not previously specified by developers. With access to user behavior information, AI can also generate optimized test cases and test data for performance testing. The same information can be used to optimize the entire testing process, as AI can predict the execution time of all tests with high accuracy, enabling efficient use of continuous integration pipelines.

5.3. AI Tools and Technologies

As the role of Artificial Intelligence in Software Engineering-specific domains continues to expand, several tools have been developed that assist or ease testers' jobs and aid domain-specific testing activities. Notable tools include MABL, Testim, Functionize, Test.AI, Appvance, and Testsigma. These tools have been the focus of detailed analyses, which consider the trends and potentials of ChatGPT and Copilot, as well as their effectiveness in software testing. Recently, ChatGPT has been scrutinized for its role in automated testing, and comparative studies have examined it alongside other tools and applications.

AI has been employed in various aspects of software testing. For example, ChatGPT has been used for test case generation and as a QA assistant. Additional studies have evaluated the performance of ChatGPT, especially during its early phases. Other Generative Pre-trained Transformer (GPT) models, such as OPT-175B, are available for developers. Copilot suggests lines of code or functions, providing insights into the appropriate formulations of input and prompts. Researchers have also proposed catalogues of ten different GPT-based utilities and explored the increasing use of the GPT model within software testing.

6. Current Challenges in Software Quality Assurance

In modern dynamic environments, where fast planning, frequent product and process releases, and constant changes are pre-conditions, a successful Quality Assurance function must respond quickly, adapt to changes, and deliver the same quality—sometimes in a stressful situation. Talk about testing early in the software life cycle (SDLC process) must begin at the business analysis level. It must consistently engage all levels of the Software Development Life Cycle so it can fully participate in the software quality discussions and calculations.

The challenge begins with the decision about the structure of the Project/Quality team members. The team members must work alongside the Project Development team members right from the beginning of the project. Solution review must become a combined effort of the developers, Solution Architects/Business Architects, and testing team members. The team needs to look at requirements from a business perspective, a usability perspective, a change management perspective, an integration and audit perspective, a technical build perspective, and from the perspective of industry standards and best practices. The team also must sense-check the Data Migration, Parallel Run, and Rollback strategy and plan. Testing scheduling must be aligned with the project development schedule.

6.1. Complexity of Modern Software Systems

The process of writing computer programs is capable of reaching extraordinary complexity. High-level programming languages and compilers now permit programmers to concentrate on the creation of complicated algorithms and data structures. Together with the advent of very high-level languages such as SQL for accessing data bases and YACC for creating simple compilers, the programming language no longer limits the complexity of modern programs. Problem complexity, however, is difficult to measure. Educational experience suggests that if the number of solutions to a particular problem is equal to the number in the Fibonacci series, the problem is interesting. The C programming language permits writing a single problem in a variety of ways: a trivial problem-solving method, a semi-complex approach, or a very intricate but fun solution.

Trivial solutions are never interesting because a simple operation and a few additions or subtractions are sufficient to build a solution. However, if the problem requires an add and a multiply and there are three possible methods for obtaining the result, then the problem is interesting. With the advent of methods such as divide and conquer, a problem can be broken first into two smaller sections, then four, then eight, etc. The complexity can be multiplied exponentially in a manner similar to the Fibonacci series. The complexity of modern software systems is therefore unexhaustible, either with regard to size or problem difficulty. Consequently, it is no longer meaningful to talk about trivial or uninteresting programs.

6.2. Integration of New Technologies

Approaches to new-support technologies differ widely in numerous respects [2,4]. Testing support has received a great deal of attention, but at one extreme there are reliance and extent-of-use issues, while at the other end of the spectrum there are dependence and suitability facets. Technologies vary from broadly enabling (but not necessarily enabling of new testing methods) to orthogonal, from enabling yet not enabling-automation to enabling and enabling-automation, from being an underlying cause of a fundamental change in the way in which testing is done (idem, for a fundamental change in testing) to not being directly considered in test-framework research, and from being mainly meant for supporting testing to mainly designed for automating testing.

Many-RE-support technologies have design-information-related content as well. Several technologies, although not explicitly mentioned in the context of designing or requirements engineering (RE), show significant interrelations with some of the core ideas/beliefs in these areas. The design-information aspect of testing-support technologies becomes especially prominent in the case of new-support technologies

associated with design, as language-agent and graphic-agent technologies. Both are designed for development support, but due to their design-orientation they have a strong influence on RE-support activities such as query formulation, analysis, evaluation, weighing-up, and discussion/correspondence.

6.3. Maintaining Test Coverage

Test coverage is key to keeping testing scalable, manageable, and relevant. The need for maintaining coverage is behind driving much of the maintenance work on the test design specification; revising it to track evolving requirements. Whenever new requirements are added, test cases are created and added to the test design specification. When requirements are changed or deleted, corresponding changes to the test cases are required, test cases will be deleted, made inactive or modified. When the requirements' traceability links are scrutinized as part of an inspection or review, inconsistencies or missing test cases are identified and additional cases are created and added to the test design specification.

Under the Test and Evaluation Master Plan (TEMP) construct, developers and the test community have some assurance that requirements test coverage will be maintained without slippage. For those requirements added, changed, or deleted, test cases must be added, changed, or deleted to correspond. Those test cases are then kept current with a current set of test requirements. The maintenance of this test case—requirements relationship is closely monitored. During integration and test phases, the lack of test cases for any system or sub-system requirement can cause this requirement to be removed from the contracting effort and requirements documentation.

6.4. Managing Test Data

Managing test data can be a major headache. Still, it is necessary if the tester is to accurately simulate the actions of a real user. A routine test might require application logins from a list of customer accounts. Other tests may need billing or shipping addresses, customer contact information, and so on. Consequently, the more realistic the test data, the greater the chance the application will be compromised during execution.

The most significant concern is the production of test data with sufficient complexity to validate the application and cross the criminal mind. Over time, the organisation may need complete test data sets for every module of every application. This is usually supplied as a pre-packaged module from the main production database. In some cases, an extract of live data may even be required for a particular component test. Of course, real data contains confidential information and is, therefore, safeguarded. However,

sensitive details can be masked with fictional data so that the live field population is maintained; the only change is the symbol and name columns.

7. Why AI is the Next Frontier in Quality Assurance

Novel technologies drive change in organizations, and the reasons for this change are clear: the new technology delivers previously impossible benefits, or it saves money.

Artificial intelligence (AI) has the potential to push the boundaries in both of these areas. The multidimensional approach enabled by AI can bring about drastic improvements in the quality and efficiency of products, services, and processes, something that has not been possible before.

7.1. Predictive Analytics in Testing

Before diving into the predictive models, a bit of context about the testing process will be presented. The testing process starts with a testing policy that can be defined using many parameters, such as, for example, the failure intensity objective, the target release date, the cost of testing, and the number of test cases for discovery. The testing policy is broken down into activities, which are defined by a valid Test Activity Record. In each of the activities, a set of tasks is executed according to the definitions in the associated Valid Test Task Record. Testing is performed by applying test cases that are defined in a valid Test Case Record. The execution of test cases yields submitted incidents that can be organized in an Incident history record (which normally looks like a three-column table with a date, the number of incidents, and the cumulative number of incidents). Indeed, this entire description is a model of the process of Software testing, albeit a semi-formal model.

A Testing Process Framework (TPF) recommends that historical incidents within the Testing Process can be used to evaluate conditions within the Testing Process; therefore, predictive models can be created by analyzing the historical incident data collected during a Testing Process in order to forecast software failure rates for the ongoing Testing Process. Hence, the predictive model emerges naturally from the historical data recorded in the Testing Process, and it can be run against the historical data to forecast testing software failure rates of the active Testing Activity. The predictive results can then be reconciled with the Testing Policy to estimate the time required to complete the Testing Process within its constraints.

7.2. Enhancing Test Efficiency with AI

As testing demands grow with increased build releases and product enhancements, conventional manual testing becomes unscalable. To control costs and compress test cycles, testing must either be minimized—risking defect escapes—or automation coverage must be broadened. Yet expanding automation demands significant time and effort, never fully achieving end-to-end testing, and still leaving many bugs undetected.

AI technologies can address these challenges by reducing manual and automation testing efforts, expanding test coverage, and identifying previously undetectable bugs. Artificial intelligence seeks to mimic human consciousness and is capable of learning from past activities, planning for anticipated events, and summarizing outcomes. AI-based tools can capture expert testers' knowledge to automate the generation of test cases and test scripts. Built upon Machine Learning, Deep Learning, Neural Networks, and Computer Vision, these learning models typically analyze multiple test executions. They anticipate future testing requirements, generate suitable test cases and scripts accordingly, and query existing executions to identify probable defect areas. These AI features can operate at any testing level—unit, API, UI, integration, system, or end-to-end—and streamline end-to-end testing processes.

7.3. AI in Continuous Testing

Continuity can give catastrophe. Consider a software product whose code is continuously changed and expanded—and thus continuously tested, too. Every test uncovers new bugs, and every new bug discovered means that more fixes will be needed. Homeostasis must be broken, which means development cannot be stopped. This feedback loop produces a stable pattern, in fact a continuous testing process.

For a continuous test pattern to work at all, the process must be automated. Automation is of paramount importance with respect to many aspects of continuous testing. One aspect is its speed: the faster a test can be performed, the more often it will be performed. The length of the feedback cycle must be reduced as much as possible, and the proper information should be provided to relevant stakeholders as soon as possible. Another aspect depends on the length of the feedback cycle—in other words, it deals with the stability of the feedback cycle. The closed loop between continuous development and continuous testing cannot be broken by interruptions, for example, when tests have been performed insufficiently because the available human resources have been insufficient.

8. Future Trends in Software Testing

Software testing is the process used to identify the correctness, completeness, and quality of developed computer software. It includes a set of activities conducted with the intent

of finding errors in software so that it could be corrected before the product is released to the user. A formal testing process begins with the execution of software requirements using several sets of test cases. The goal of software testing is to see whether the actual behavior of software matches the expected behavior which the actual software has been produced for.

Throughout their lifetime, software programs have been subjected to many changes caused by corrections and new functional requirements; these changes create the necessity to check the program's correct functioning periodically and automatically, in order to reduce time and the human effort required. Software testing has been around as long as software development. In the beginning, software was produced at a slower pace with small teams, so manual testing methods were sufficient to ensure quality. Since there was a requirement to test the contemporary program, it was other testing methods, such as exploratory testing, which injected the human into the lifecycle of testing. But today, software is written very quickly and maintained by much larger teams. Consequently, manual testing methods cannot keep up with the increasing pace of software development and maintenance. Modern software development processes require automated tests run for every new build. This yielded a new era of automated testing tools, which make the execution of thousands of tests possible in hours or minutes, as opposed to weeks or months. Two different software testing approaches derived from the development process: manual and automated testing approaches.

Despite the fact that automated tests execute faster than manual ones and as time passes the number of automated tests grows, creating and maintaining them remains expensive. More recent research in the field of software testing now combines AI techniques with conventional testing strategies in an attempt to improve efficacy. MIRATA is an experiment that explores the application of search based software testing and machine learning techniques in a production environment. One of the primary reasons why artificial intelligence is being adopted to augment software quality assurance is the sheer number of challenges that quality assurance is facing today, such as: handling massive integrations, testing on multiple platforms, testing during ongoing development, and regression testing. Artificial intelligence’s ability to predict code success/failures and cluster “like-type” test cases for better understanding on coverage and detection also help address many of quality assurance’s challenges, making artificial intelligence the next frontier in quality assurance.

9. Case Studies of Successful AI Implementation

AI has made a significant impact on the field of software testing. Landmark AI-based testing frameworks, such as EvoSuite, Randoop Test Case Generation, and ML Suite, have made their mark. Next-generation test automation driven by AI reshapes the QA

ecosystem, delivering faster releases and reducing costs. Most companies have chosen to use AI in testing, and AI-assisted testing tools reduce redundancy in testing. Providing a bird's-eye view of the end-user experience, AI practices lower the time, cost, and complexity of Ad hoc testing.

AI-integrated testing is advantageous, helping to achieve more with less capital and time. It enhances software testing through algorithmic implementation that eliminates redundant manual checks and processes, transforming software in a sustainable way. Testing provides suitable inputs for training machines.

Quality Review and Assurance (QA) is an inherent part of the software development restoration lifecycle. Manual quality assurance is time-consuming and requires assigning special testers or developer resources for QA. Testing automation comes to the rescue, offering faster testing, faster feedback to developers, and more repeatable and effective testing.

10. Best Practices for Implementing AI in Testing

The adoption of AI-based testing requires a different approach compared to the classic one. To do that, organizations should consider several best practices for successful implementation of AI in testing:

- Clearly identify and state the business problems to be solved or the process areas to be transformed - Determine expected benefits and outcomes of the AI implementation - Examine and analyze the current state of applications and test processes - Conduct proof-of-concept and pilots, considering the lab environment and the AI training/data models - Incorporate the lessons learned into the strategy for scaling up the implementation

Implementing AI in testing is a journey that can span several months or even years. It is important to understand the current capabilities of AI-based testing tools, the challenges faced, and how to overcome them. One must also look ahead to what the future of AI in testing holds and the dynamic side of AI implementation.

11. Ethical Considerations in AI Testing

New challenges in AI testing are mostly social in nature, as developers tend to build and test AI applications for narrow scenarios that can be limited by the development environment. Deploying AI applications in different contexts can raise ethical issues of law, privacy, adverse effects, and societal bias.

As more decisions by AI now affect human lives, it is becoming crucial to consider these implications as a part of testing such systems. The term “. Ethics in artificial intelligence" is usually attributed to Alan Turing who advocated for the inclusion of ethics in later

stages of AI development (e.g., Asimov's three laws). More recently, focus has been on ensuring that an AI application does not unfairly discriminate against specific social groups. In practice, this usually means testing machine learning models for variations in equal-opportunity accuracy.

12. Conclusion

Quality control is no longer simply a stage in software development, it has, instead, become an entire subarea of software engineering. The growth in quality control has created specialized teams of engineers who test and validate new computer applications. Quality control sometimes is treated as an engineering specialty distinct from the specification, development, and maintenance of new programs.

PCs have become extremely complicated. They are no longer just computational workstations for personal or business use. Software testing now extends beyond the individual program. It involves integration testing, performance testing, batch scheduling, job scheduling, and so on. An application program is being treated as a simple subunit of a much greater super unit. Although these testing processes have only recently been proposed, they have already gained strong acceptance and play an important part in the quality-control process for all major areas using business applications.

References

[1] Kuhn R, Kacker R, Lei Y, Hunter J. Combinatorial software testing. Computer. 2009 Aug 7;42(8):94-6.

[2] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.

[3] Sneha K, Malle GM. Research on software testing techniques and software automation testing tools. In2017 international conference on energy, communication, data analytics and soft computing (ICECDS) 2017 Aug 1 (pp. 77-81). IEEE.

[4] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.

[5] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.

[6] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.

[7] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.

[8] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).

[9] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.

[10] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.

[11] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.

[12] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.

[13] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.

[14] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.

[15] Talby D, Keren A, Hazzan O, Dubinsky Y. Agile software testing in a large-scale project. IEEE software. 2006 Jul 17;23(4):30-7.

[16] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.

[17] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.

[18] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.

[19] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.

[20] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.

[21] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.

[22] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.

[23] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.

[24] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.
[25] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.
[26] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.
[27] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.
[28] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[29] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.
[30] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[31] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.
[32] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.
[33] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.
[34] Harman M, Jia Y, Zhang Y. Achievements, open problems and challenges for search based software testing. In2015 IEEE 8th international conference on software testing, verification and validation (ICST) 2015 Apr 13 (pp. 1-12). IEEE.
[35] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.
[36] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.
[37] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.
[38] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.
[39] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.
[40] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.

[41] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.
[42] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.
[43] Mäntylä MV, Adams B, Khomh F, Engström E, Petersen K. On rapid releases and software testing: a case study and a semi-systematic literature review. Empirical Software Engineering. 2015 Oct;20(5):1384-425.
[44] Garousi V, Mäntylä MV. A systematic literature review of literature reviews in software testing. Information and Software Technology. 2016 Dec 1;80:195-216.

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 2: Artificial Intelligence and Machine Learning for Test Engineers: Concepts in Software Quality Assurance

Partha Mohapatra

AT&T Corporation

1. Introduction

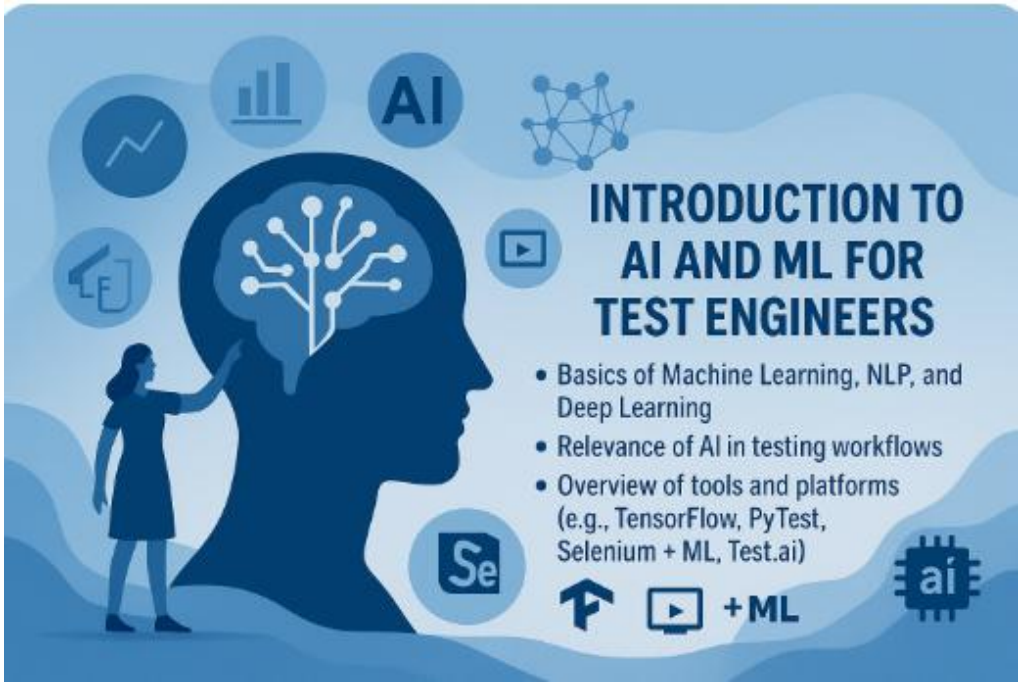
Artificial Intelligence (AI), Machine Learning (ML), Deep Learning (DL), and Natural Language Processing (NLP) are widely discussed in software engineering and testing due to the tangible benefits they offer [1-2]. While AI terminology can sometimes mislead, ML is at the core of many AI-related tasks; therefore, ML discussions receive significant attention under the AI umbrella [2-4]. These fields are not only trendy but also highly relevant to Application Development Services, Operational Automation, and Test Experts [5-6]. In software testing, AI-related models are implemented to create predictive analytics, generate test cases, and establish comprehensive automation-supported test frameworks.

AI, ML, Deep Learning, and NLP form interrelated components of the automation sphere in software development [7,8]. Selecting a suitable tool from the plethora of available options requires an understanding of the testing workflow, the relevance of each AI component to practical testing scenarios, and an awareness of the skillsets necessary to operate these tools effectively. The existence of numerous open-source, semi-open, and commercial tools demands a comparative and more generic perspective to benefit the test automation community. This paper aims to distill the basics of ML, NLP, and Deep Learning, explore their applicability in the testing domain, identify relevant tools for test engineers, and provide insights for implementing AI at various levels of the test framework to enhance time-to-market and reduce delay risks.

2. Basics of Machine Learning

Though test engineers may collaborate with AI/ML solutions, the AI/ML scientists and engineers themselves are the source- or application-data specialists in the solution environment [9-12]. As some have said, AI/ML is a "teach a person to fish" proverb—not a robot-fish-bot that catches the fish for you. Training and testing the AI/ML solution takes specialist people using the specialist tool suite, in the specialist process and environment.

ML enables AI learning from experience without being explicitly programmed, building mathematical models based on sample data or past experience [7,13-15]. Generally, it is used to make predictions or classification based on labeled training data. Using a model built with supervised ML, suggested defect and test case priority may be provided. Unsupervised ML models use unlabeled sample data sets, operate on data without a classification label, perform clustering, determine patterns in large data sets, and segment the data sets into clusters.



2.1. Definition and Concepts

Machine learning aims to develop systems capable of "learning" and adapting. It is the study of computer algorithms that improve automatically through experience. More formally, as stated by Arthur Samuel, machine learning gives computers the ability to "learn without being explicitly programmed." Tom Mitchell proposed a similar

definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." Supervised learning and classification refer to the process in which a machine learning algorithm learns from a training set of labelled data instances—examples that have classes associated with them—then later classifies unseen instances of the same form.

Supervised learning related to prediction of continuous values—regression—consists of estimating, forecasting, and predicting a value for some credential [9,16-18]. Some examples include predicting the price of a stock for the next day, the value of currency in the future, the value of CO2 emissions in the United States, and the population of a specific country a few years from now. Supervised learning related to prediction of categorical values—classification—consists of predicting a label or category that a data instance belongs to. Some examples include the prediction of the gesture type a person is making, the identification of a printed digit, and the recognition of a writing style. Prediction of discrete future values is known as sequencing. Some examples include predicting the movement of traffic lights, electricity demands, and weather status.

2.2. Types of Machine Learning

A test engineer also should be aware of the types of ML learning, to decide when supervised learning or unsupervised learning could be used. Supervised learning is a method of machine learning where the machine is "trained" using well-labelled datasets, containing a set of inputs and their corresponding correct outputs. Supervised learning is used to solve classification and regression problems [2,19-20]. Classification is about predicting discrete outputs, while regression predicts continuous values. Some common algorithms for supervised learning include Linear Regression, Logistic Regression, Support Vector Machines, Decision Trees, Random Forest, and Neural Networks.

Unsupervised learning consists of machines learning from datasets with unlabelled data, i.e., no labels or answers attached, hence no guidance or feedback is provided to the algorithm during the learning process. Unsupervised learning is typically used to solve clustering problems, where the goal is to group data items with similar characteristics. Some common algorithms for unsupervised learning include K-Means, Hierarchical Clustering, DBSCAN, and Association Rule Learning. Neural Networks are also used for unsupervised learning in the form of autoencoders and Generative Adversarial Networks.

2.3. Common Algorithms

Machine Learning (ML), Artificial Intelligence (AI), and Deep Learning (DL): What are the differences and in what application domain should I use which approach? Starting with an unstructured data set or a complex process, the easiest or most obvious solution is usually to use Artificial Intelligence (AI) [9,21-23]. Explaining it can be difficult because of the unclear or fuzzy borders between ML, AI, DL, and others. These terms are used differently depending on the user's knowledge and background.

Artificial Intelligence (AI) is the science of making machines smart and intelligent, allowing them to take actions to perform decision-making tasks. The goal of AI is to stimulate the working of human brains such as reasoning, learning, and planning. Machines can perform these tasks independently. However, it is impossible to use only AI without Machine Learning (ML) and Deep Learning (DL). These technologies make it possible to perform AI tasks through AI training.

ML performs the training of data, and DL is used for processing the training data. ML is software that builds machines that are capable of learning, understanding, reasoning, and performing like humans using AI principles [24-26]. It is defined as the study of computer algorithms that improve automatically through experience using statistical techniques. DL is a new and advanced area of ML. It is a powerful function provided by ML for automatic learning and feature detection from data using the artificial neural network. It is made of more hidden layers to solve complex real-world problems.

In the context of test systems, AI and ML technologies within test equipment allow system components to collect, analyze, and act on data according to identified trends or anomalies. This enables testing systems to become proactive rather than reactive to change.

3. Natural Language Processing (NLP)

Natural Language Processing (NLP) is the subfield of artificial intelligence (AI) that focuses on designing, developing, and deploying algorithms, methods, and systems to provide machines with the ability to comprehend, analyze, and derive meaning from textual or linguistic input [8,27-30]. Natural Language Understanding (NLU) is a subset of NLP concerned with tasks such as categorizing texts, recognizing entities and feelings, summarizing content, and detecting relationships. NLU also includes activities like parsing, part-of-speech tagging, dependency analysis, and coreference resolution.

Natural Language Generation (NLG) is the complementary domain of NLP that focuses on generating text from various information sources and in different languages [9,31-33]. NLG encompasses applications such as answering questions in a dialog, composing

emails, generating summaries, creating reports, describing images or charts, translating documents, explaining decisions, and summarizing meeting minutes. The proper use of grammars, styles, tones, and formats all fall within the scope of NLG.

3.1. Introduction to NLP

Natural language processing, or NLP, refers to a broad concept in the greater domain of artificial intelligence, or AI, and refers to all things related to the communication and processing of natural languages by computers [34-36]. Readers may hear as well natural language understanding and natural language generation, which are considered subdomains or subsets of NLP, but the fundamental definition of NLP covers all natural language processing. NLP is all about human interaction with computers and the subsequent interpretation and processing of that communication. As such, NLP includes written and, therefore, digital texts but also some speech elements; these are inbound communications "read" or "heard" by a computer either for immediate response or some form of processing.

Test planning and design involve a lot of communication, so much so that two of the work products produced in the process are Test Plans and Test Designs, or simply Requirements and Description. These texts contain a level of detail necessary to successfully test the system under test and decrease ambiguity. For example, a Test Plan would describe in detail what is being tested and why, such as a description of the individual tests, requirements traceability, schedule, resources, and risks involved. A Test Design, on the other hand, details the coverage of the system under test, such as which requirements will be tested and which techniques, methods, and level of coverage will be used. The content of Test Design is normally traced back to the Test Plan through references, to show how it contributes the overall achievement of a particular Test Plan objective. The Test Planning and Design phases of the software or system development lifecycle can be viewed as a communication loop between the test engineers and the requirements and design stakeholders.

3.2. Applications of NLP in Testing

Tool support is the key to enable a DevTest transformation and is crucial in effectively implementing processes such as continuous and shifting left. Changes in process and the increased demand on resource skills including DevTest engineers, open an avenue for AI and ML to contribute towards software testing, assisting with both efficiency and effectiveness. Natural language processing (NLP) is one of the AI/ML areas which offers promise for the area of software testing. Test artefacts such as user stories, requirements and acceptance criteria are written in natural language form. An intelligent test solution

that understands, uses and creates natural language can ultimately democratise all software-testing phases. Using only the test conditions in their natural language forms, test engineers will be able to automatically generate various test artefacts across the software-testing phases (test-design, test-oracle, test-execution and bug-logging phases). NLP has been on the radar of software testers for a long time and the results presented in the academic literature have demonstrated the potential for implementation within some test phases.

Testing is the backbone of great software products and hence developers and testers know code and testing well. Creating tools that can interpret natural language test conditions, generate the tests and execute the relevant tests for the developers/testers will ease their workloads and push the release velocity. Natural language problems are difficult for computers and if developers/testers have to create a checklist of all the test conditions (test design) in natural language and ask the tool to run it, this exercise will be tiresome without any advantages. Inspecting the natural language descriptions of the features (written in the behaviour-driven development format) enables a tool to generate any information related to test design, test oracle, test execution and bug logging, within the testing phases. NLP enables the latest tools to extract the relevant information and generate test artefacts across the software-testing phases without requiring any external setup, such as data sources, configurations and additional inputs from developers/testers.

3.3. Challenges in NLP

Although neural network techniques have been enormously successful in various machine learning applications, certain aspects of natural language continue to challenge neural networks [3,37-39]. Most algorithms work well, provided that linguistic elements can be represented as vectors in metric space; for example, word embeddings, sentence embeddings and sentence transformers take words or sentences and represent them as vectors of real numbers. While much of the linguistic information in natural language can be represented in this way, certain elements remain challenging to model as vectors in Euclidean metric space—for example, antonymy (bank:river opposite of bank:money); hypernymy (dog (hyponym) is animal (hypernym)); and meronymy (door is part of a house). Other difficulties include handling language ambiguity, complexity, context-dependence, and synonymy.

Hyperbolic spaces provide an alternative method for representing embeddings, which naturally align with the hierarchical nature of hypernymy relations. For example, in a hyperboloid model of hyperbolic geometry, animal is at the centre of a two-dimensional space, with dog, cat and lion situated some distance from it, and further orders of subtypes are arranged proportionally at increasing distances from the centre. Such a configuration cannot be achieved in Euclidean space but is very natural in a hyperbolic

space. This approach was illustrated by Riemer, who mapped a hypernymy hierarchy for animals by embedding a Gaussian hierarchy mapping into hyperbolic space.

4. Deep Learning

Deep learning is a subset of machine learning that uses a class of models called deep artificial neural networks that are inspired by the structure and function of the brain and that aim to emulate brain functioning in a very simplified manner [36,40-41]. For many years, using deep ANNs on a much larger scale than ever thought was possible was not feasible, due to their hardware requirements. But since around 2006—with the advent of Graphics Processing Units (GPUs) and lower memory costs—the technology has skyrocketed and increased in popularity. The breakthrough came when Geoffrey Hinton and colleagues showed that it was possible to begin stacking thousands of neurons in neural networks and train them layer by layer. As a result, deep learning has been implemented across many fields and has become the dominant approach for difficult tasks such as image classification, object recognition, speech recognition, and machine translation.

The deep learning framework refers to the tools that support the design, training, and validation of deep neural networks. Some of the most popular frameworks include TensorFlow, PyTorch, Theano, and Caffe.

4.1. Understanding Neural Networks

Fundamental to much of the progress in AI in recent decades has been a technique known as neural networks, which provide a powerful method for learning patterns within data. Neural networks are based on graphs containing weighted nodes arranged in layers. Inputs to the model are presented to nodes in the first or an early layer, and weighted sums of node values are propagated forward through the graph until each node in the final layer emits a network output. The model's answers are compared to expected targets and the difference used to calculate an error value, which is then propagated backward through the network to update the weights. This approach is known as backpropagation for error correction. After repeated rounds of data presentation and weight updating, the network can perform tasks automat ically, such as classifying text documents, assigning topics to sections of code, or even generating plausible completions to those sections. These models excel at pattern separation and relationship extraction in data, rather than higher-level analytical or reasoning tasks.

Neural networks are employed in operations such as verification and marketing classification but also in generating text and image completions. In AI, they form part of

large language models, such as ChatGPT, for instance. Neural networks are also essential in cognitive building blocks for textual search enrichment and are extensively utilized in markets for AngelHack Titans and BlackHat Brain.

4.2. Applications in Software Testing

After almost seventy years of AI history, it is evident that AI is not a single invention, but rather an umbrella term for a range of applications designed to make self-driving cars, chatbots, robotic kitchen helpers, and much more. Each activity uses a specific form of AI best suited to the task. For instance, a self-driving car traveling both around supermarkets and in the countryside relies on neural networks and computer vision to recognize the environment and adapt its behavior accordingly.

Applications that are licensed to require Extensive AI Some of the applications include predictive coding, sales forecasting, and automated driving of vehicles—even at low speeds, automated purchase and sale plans in manufacturing configurations, and robot arms.

4.3. Tools for Deep Learning

Deep Learning focuses on training artificial neural networks with multiple hidden layers. Deep learning tools provide the ability to define neural network architectures by assembling required layers or components in a graphical interface, making it reliable for visualizing the expected network's purpose [9-12].

The main purpose is to build a deep learning network by assembling pre-available layers with specified parameters. A layer, such as a 2D convolutional layer, requires specific parameters like the number of filters, filter size, stride, border mode, and padding. Tools often provide instant visualization on how the shapes of inputs/outputs change after applying the layer, enabling the design of meaningful end-to-end convolution layers by stacking them. Convolutional layers play a crucial role in making the network applicable to images and performing feature extraction as they operate on local regions.

5. Relevance of AI in Testing Workflows

Integration of Artificial Intelligence (AI) into the core of business functions causes it to permeate every aspect of a company’s operation. Therefore, even seemingly unrelated tasks grow influenced—especially those that rely on data, business knowledge, and adaptability—enabling the use of tools for higher productivity. In testing, machine learning (ML) applications are varied: organizing test tasks, generating bug reports,

assisting in incident reporting, and automating GUI testing, among others. Testing departments can leverage machine learning techniques for Choosing the next test case, Predicting the pass or fail of defect reports, Assessing the risk level of defect reports, and Assigning defect reports to the right developers. Whether these techniques yield acceptable error rates is the key inquiry.

Modern Web applications employ Javascript libraries and services that deliver fluid, component-based, and decoupled user experiences. They take the form of stateful engines that continuously trigger events, often resulting in a few changes on the browser Source Document Object Model (DOM) (for example, toggling the display property of some HTML element), thereby making classic event-based representation obsolete. Testing such complex applications requires understanding and building page state models (PSMs). An approach targets the automatic construction of event-level PSMs for complex stateful AJAX applications (such as Trello). The primary purpose of the PSM is to aid replayer agents that test such applications.

5.1. Enhancing Test Automation

The role of AI and ML in test automation is becoming increasingly important because testers need to improve the test efficiency and ensure quality while testing complex, large applications with reduced time and effort. Test automation has become a necessity under these considerations. AI and ML support this by designing intelligent, smart, and adaptive test automation programs in addition to reducing the amount of manual intervention needed throughout the development lifecycle.

Test automation consists of the design of automated test scripts and their execution on the target application. The design of automated test scripts for applications based on modern technologies is itself a challenge. Adapting existing tests for new changes, especially for large and complex applications, is costly. Once the test scripts are designed, their updated versions need to be executed periodically, as mentioned earlier, which is time consuming and expensive. Prioritization of the test scripts for execution can mitigate the excessive costs and time requirements. The evaluation of results generated after the execution of automated test scripts is also crucial, since it should be fast and accurate. AI and ML approaches can support all these test automation activities by delivering intelligent bug detection and by providing smart analysis of the test results.

5.2. Predictive Analytics in Testing

Predictive Analytics entails harnessing the power of Artificial Intelligence and Machine Learning-based recommender systems to identify components that are most vulnerable

to defects. Comprising two major phases, it focuses on either the specifications or the source code. In the first phase, recommender systems are employed to locate features that are likely to be defective. In the second phase, they predict developers who are predisposed to introduce bugs into the source code.

Three types of prediction tasks exist: feature-based defect prediction, source code-based defect prediction, and developer-based defect prediction. Prior to selecting the prediction model, an extensive set of features from industrial software projects must be extracted and organized into a dataset. Signal analysis operators serve as a valuable means of automatically identifying features in specification documents.

5.3. AI-Driven Test Case Generation

Machine learning, a subset of artificial intelligence, is designed to autonomously detect data patterns and glean insights without human intervention. Initially, training datasets were modestly sized, enabling classical computer systems to perform rule-based analyses. The advent of big data and high-performance computing facilitated a progression towards advanced GPU-based learning models, propelling the AI frontier beyond rudimentary pattern recognition.

Deep learning propelled numerous AI applications, many of which now incorporate aspects of artificial natural intelligence. The gaming domain achieved substantial milestones, with AI-driven bots winning chess matches against top-business grandmasters and conquering all levels of Moorhuhn. The evolution of bots, transitioning from expert systems to mimics of human cognition, underscores the transformative trajectory of the field.

6. Overview of Tools and Platforms

In recent years, numerous modeling frameworks and development kits have been introduced to increase the usability and applicability of artificial intelligence (AI) and machine learning (ML) technologies. Most of these tools are open source; they are supported by stable open communities that ensure bug fixing, development of new features, and continuous upgrades compatible with recent improvements in AI/ML algorithms. These frameworks also provide manuals describing their APIs and the underlying theory. For test engineers wishing to build small models for simple experiments or exploration activities, open source AI/ML frameworks with Python bindings are recommended. This section explores several popular toolkits and frameworks for such tasks.

Open source frameworks are freely available but have associated maintenance costs. Cloud services offer ML, deep learning (DL), natural language processing, speech processing, and computer vision services with low operating costs; they charge customers by the hour and provide machine models at various classification levels. Cloud services support early exploration and market development for businesses but may reduce control over core business capabilities and customer data. Moving away from cloud services can result in heavy sunk costs. Cloud service providers do not significantly differ in pricing; instead, buyers should consider vendor-specific service catalogs, service level agreements (SLAs), service quality, and privacy policies. Some of the best-known cloud vendors providing ML services include Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure.

6.1. TensorFlow

TensorFlow is an open source framework designed for machine learning and numerical computation. It is backed by Google and the TensorFlow Extended (TFX) platform for practical ML production pipelines. The TFX platform is based on Apache Beam and allows training models in the cloud or on-premises. The TFX infrastructure consists of libraries and tools that enable the deployment of ML pipelines.

TensorFlow is not a machine learning library per se; rather, it is a low-level library of base algorithms that allows engineers to build the advanced architecture. Bear in mind that each complex library is based on low-level algorithms, and TensorFlow certainly fits the bill.

6.2. PyTest

pytest is a popular Python testing framework used for unit, functional, and acceptance testing. Key features include simple tests, scalable fixtures, parameterized testing, unittest support, pytest.ini for customization, and straightforward setup.

The pytest fixture API permits test parametrization and modular markers. A test can be started in multiple forms by utilizing additional inputs via decorators: @pytest.mark.parametrize according to data, @pytest.mark.webtest according to type, @pytest.mark.skipif for conditional cases. Running tests in parallel allows tests with @pytest.mark.webtest to execute simultaneously on different machines.

6.3. Selenium + ML

Selenium WebDriver comprises a set of web application development testing tools, but it does not account for any machine learning. A Gray box model is required for evaluating user behavior. Imagine Selenium as a framework for an enterprise and proper models for user behavior trained on datasets such as e-commerce. Selenium WebDriver produces different test cases to test the web application. The data collected are user actions, timing spent on the web page, number of users on the website, etc. Third-party tools like Countly, MixPanel, and Pendo generate similar reports about the web application for its users. Selenium's limitation is its inability to predict the test case priority for execution in Regression Testing. A Machine Learning model utilizing the K Means clustering algorithm and the SVM classifier can find an optimal solution for this limitation.

The collected data are split into two categories: training data and testing data. K Means clustering classifies the data into new and old clusters. A Support Vector Machine styles new clusters as New Test Case Priority and old clusters as Old Test Case Priority. Finally, the various test case priorities are executed accordingly. The same process is attempted with IBM App Scan in conjunction with the K Means clustering algorithm for vulnerability testing. Testing a website for functional bugs—for example, watching its behavior when adding additional items to the shopping cart in Flipkart.com—is Like or Dislike analysis, which can be done with the aid of Selenium and Artificial Neural Network. An approach using the rate of change of area indicates whether the Add to Cart button is Change/No Change.

6.4. Test.ai

Test.ai performs AI-powered testing of mobile applications on iOS and Android. The tool comes with pre-built tests and functionality based on typical AI-derived user flows for mobile applications, for instance, onboarding and login. It is useful when wanting to test an extensive range of possible user flows, including checking application UI, functionality, and behavior. Test.ai employs AI to write tests, utilizing AI models and following common commands for mobile-app test automation. The tool identifies on-screen objects that support smart, easily scalable test construction. The infrastructure supports tests across multiple domain areas, such as watches, voice, and chat.

6.5. Comparative Analysis of Tools

The following tables provide a comparison of ML libraries and tools relevant for implementation in automated testing. Table 6.11 contrasts Google’s TensorFlow with

Scikit-learn and Amazon SageMaker. TensorFlow supports many languages and a very broad range of models and platform applications. Scikit-learn, a Python toolkit, provides a limited number of algorithms at a much simpler level. SageMaker offers robust functionality as a Cloud Service Platform. Table 6.12 provides a summary of features for CPI Demon and Test.AI, which are developer-oriented test tools that use AI and ML concepts. CPI Demon is very tightly integrated with the Jira requirements-management tool, while Test.AI offers a much broader coverage and higher level of automation.

7. Best Practices for Implementing AI in Testing

Six key best practices will enable testing teams to navigate AI and ML adoption challenges and achieve success. First, testers should evaluate their organizational readiness by assessing their skills, stability, and tolerance for AI's messiness. Pilot exercises can help determine the likelihood of sustaining AI testing initiatives. Second, the focus must be on reducing the cost of testing, rather than simply increasing automation. Identifying the most onerous and costly testing activities will yield the highest resistance to failure and ensure optimally allocated investments in AI.

Third, intelligent code coverage analysis that continually learns from the business context presents a significant opportunity for cost reduction. Fourth, transitioning to visual automation frameworks scales easily and enables the automation of GUI test cases across front-end platforms. Fifth, implementing AI-assisted testing workflows standardizes testing across the organization and can guide testers of all skill levels and experiences through the testing lifecycle. Visual assistance in executing complex testing processes reduces dependency on specialized skills. Finally, the growing availability of open-source AI frameworks and ML services demands a critical internal evaluation prior to investment. Deploying AI frameworks may require not only technical skills but also skilled program managers and testers with data science backgrounds to achieve the desired outcomes.

7.1. Choosing the Right Tools

The topic of which machine learning model is the best has been asked hundreds of thousands of times. The correct answer is that it depends on the job — in fact, it’s less about the class of the model and more about the specific data set itself. The tools used to build the model are typically not that important, even though you might hear talk about how XGBoost, LightGBM, and CatBoost are all current state-of-the-art techniques.

Many open source communities, volunteer projects, and well-meaning individuals have created thousands of automated machine-learning libraries, ranging from poorly

documented to A+ levels of polish. These frameworks commonly contain support for every class of model, including neural networks. The major differences between the various AutoML frameworks can usually be boiled down to one or two of the following categories:

- ease of setup and configuration - the ability to naturally handle certain data types such as images and text - available preprocessing options - amount of automated feature engineering - the amount of model tuning allowing for different types of models to perform better - customization options - execution speed

7.2. Integrating AI with Existing Workflows

Integration of artificial intelligence (AI) into existing continuous integration (CI) and testing workflows can be approached as a continuous safety net, configured to operate alongside current setups without requiring immediate coverage of all corner cases. Inevitably, the introduction of AI brings a transient period marked by noise and false alerts. In many settings, dealing with such noise is less problematic than encountering undetected issues at the outset of the journey. When users understand tuning options and the system's mechanisms, a gradual reduction of this noise becomes achievable. Furthermore, the presence of myriad false alerts serves as an indicator that certain components are misaligned with the system's statistical expectations.

Continuous integration offers an excellent avenue for this pilot phase of machine learning (ML). Each build operates on an identical codebase but remains distinct regarding specific inputs and outputs. Leveraging ML to produce unique insights for each build means that the majority of computations are not prone to repetition. Utilizing the union of sets across all tests enables the system to learn, thus eliminating the need to repeatedly perform millions of identical actions. Although this entails costs on the server side, the manifest advantages to the workflow, along with the freedom to analyze any build in another context, provide ample justification. Additional benefits include the capacity to compare upward trends and deviations over time within identical tests and the automated generation of comprehensive reports when it is known—whether through internal monitoring or user discussion—that unit tests are failing for particular reasons. The user is simply required to ascribe a label to these reasons.

7.3. Training and Skill Development

The evolution of test-driven development demanded the automation of routine testing operations in web and mobile projects. Nevertheless, the transition toward automation often led to profile shifts in interdisciplinary teams. Consequently, test engineers should capitalize on their business and domain expertise by pursuing Artificial Intelligence and

Machine Learning (AI and ML) training. Extensive, specialized training ensures that their skill sets remain both relevant and future-proof.

Several institutions provide foundational AI programming courses taught by leading scientists, presenting an ideal opportunity for test engineers to acquire practical skills in Data Science, AI, Deep Learning, and Machine Learning. Mastery of these programming and technology skills will facilitate the creation and maintenance of AI models trained to understand the target application and business. Properly trained AI models enable automation of mundane testing tasks.

8. Future Trends in AI and ML for Testing

Test Automation for Production Releases Test automation is a gamechanger, but its value varies across the software development process. Test-ers conduct Production Release Automation that involves testing with real data under real operation in the real Production environment. Testing approaches for Simulation-Driven Production Release Automation (SDPRA) can be classified based on whether a test focuses on the Simulation phase or the Production environment. AI and ML methods are employed to automate the process in both contexts.

Test Automation for Early Software Development Stages The benefits of automation are most significant during the conception-to-releaseâ€ ” or design-to-releaseâ€ ” software stages for AI and ML projects. During the early stages: Test teams build AI- and ML-based automatic tools with VIPER (Verify Inner Features Personalities of the Engineered Reality). Inner features refer to features that stimulate the engine to make decisions; these features represent the engine's personality, or character traits. Engineers examine the excitatory and inhibitory features of the engine, which—in a psychological analogy—induce "happy," "sad," "rage," and "fear" feelings. SANDI (SAnding Neural DepItion) is a Natural Language Processing–based technology for futuristic embedded intelligent testing. SANDI research has been conducted in conjunction with Thales Research & Technology, focusing on the translation of natural language technical specification documents into suite-specific source code in the Python programming language.

8.1. Emerging Technologies

Data processing at scale has become both a Loader and Battery for AI development. The speed of transforming masses of data into decision-making information is the primary difficulty. Traditional databases and Enterprise Data Warehouses (EDWs) were not designed to keep up with the massive amounts of data generated by pipeline activities.

In the last decade, the technological solutions developed, known as Big Data and Data Lakes, at least meet the requirements that everyone expected for data processing together: volume, velocity, and variety.

Big Data and Analytics are at the center of attention of major corporate organizations due to the new business opportunities offered by correct information use in combination with traditional data sources. The mobile and IoT wave of adoption is letting firms collect large amounts of data, while advanced analytics can extract valuable information and allow the connections among all real-time generated events. Moreover, artificial intelligence capabilities promise to change the whole data ecosystem.

8.2. Predictions for the Next Decade

Predictions for the Next Decade Regardless of what one thinks of generative AI today,engineering and society will be shaped by it in the coming decade.Whether it will be remembered as the AI Winter, or the Pyrrhic victory forhumanity, like that of superpowers with nuclear weapons, remains to be seen.Some researchers have a more balanced and measured view, for instance,Albert Einstein, in a letter to a friend in 1932, noted: “The time willcome when man will be able to use a machine with the aid of which he willbe able to put in a few seconds the knowledge and experience of all mankindthat has in the course of centuries been created and stored up.” What he didnot say, yet what was implicit, is that how mankind uses and controls thatknowledge will also matter – something that is being overlooked today.

Time and again, researchers have predicted that every couple of yearsmachines will be able to do all kinds of tasks that today only trained humanscan perform. Nevertheless, given the current pace of technology, the challengeof safeguarding humanity and the planet will soon move out of the realm ofspeculative fiction with the realization of advanced AI. Responsibility thenshould belong to AI researchers, their funders, governments, and all whomay ultimately decide the usage of the new technology. Two examples are Work titled Introduction to AI and ML for Test Engineers

Take the Role Instead of Ending Up Being Replacedby the Machines by Pranaya Mishra and Software Testing AI: 25 Predictions forSoftware Testing in 2024 Impact of ChatGPT on Testing Industry by Testing9.com.

9. Case Studies

Widely recognized Squashed Heads (SH) glitches in video games occur when the characters' heads become out of scale with their bodies — either too small or too big. A compelling test sequence for AI and ML specialists reviews classic and modern video

game artifacts together with correlation-based explanations derived from a Siamese neural network. It is relevant to specialists who build frameworks for automated game testing.

A Siamese neural network predicts the degree of visual similarity among reference and intentionally squashed heads, and establishes correlation with respect to key behavioral indicators. Popa, Tudose, and Barlea show that changes in the visual representation of sprites are associated with decreases in subjective ratings. While the analysis of visual aspects in video games is challenging, the proposed deep-learning approach delivers sensible recommendations. Introducing a new class of artifacts and a corresponding distance metric encourages future investigation and expands the possibilities of automated game testing.

9.1. Successful Implementations

Artificial intelligence (AI) and machine learning (ML) are two of the newest approaches to modernizing testing of software and complex systems. Nonetheless, more experienced practitioners see them as the newest "bandwagon" that is here today and will be gone tomorrow. Still other practitioners welcome them for helping solve some of the current challenges in AI and ML.

These new approaches address two major problems: (1) too many test artifacts and (2) constant and complex changes in software and its operational environment. Traditionally, software testing is a very human-intensive process—because of these challenges, it remains that way even in the modern world. The so-called smart testing methods purport to make testing more intelligent by automating it, thereby reducing the human involvement and making testing more capable in dealing with change. Even with all these promises, the following paragraphs point out actual implementations of AI and ML in recent years in the larger context of software engineering and its related fields.

9.2. Lessons Learned from Failures

AI and ML have shown great promise in routines that aim to search the seemingly infinite test input space. However, many interesting ideas reported in academic papers have not delivered on the promise. Yet, attempting to use AI or ML in testing is never a wasted effort, since valuable experience can be extracted from failures. Many failures are the result of poor understanding of AI and ML principles in the software-testing community, but there are also failures caused by the unreasonable expectations that AI or ML can help in testing even for low-level, highly automated test activities.

For example, AI and ML should not be expected to help in generating boundary-value conditions or in functional testing with well-defined input/output methods. Indeed, ML tends to be data-hungry. During failure analysis, it is important to ask the question "does the testing problem have significant amount of quality test-data?" For example, to do AI-based test Triage for the features of a product under test, one must have appropriate data containing thousands of bugs and feature requests. Similarly, AI-based Test-Case Prioritization requires historical data from many previous test-execution cycles.

10. Conclusion

As Artificial Intelligence (AI) and Machine Learning (ML) continue to gain momentum in both the world of technology and society, they proliferate into an increasing number of areas of life. Testing is a prime example. The field of testing has seen many changes over the last few years, with several companies developing intelligent test solutions. Smart test solution frameworks have been enabled through various technological advances involving AI and ML. Undoubtedly, these kinds of smart solutions have many benefits for the testing discipline. However, it is essential to carefully understand, question, and check the very core of those intelligent or smart systems before deciding to implement them. This is because these applications are powered by AI and ML, which themselves are driven by data.

Therefore, it is perhaps necessary to take a very fundamental look at how AI and ML operate, examine their architecture, and appreciate how these domains can be beneficial not only in the area of testing but also to test engineers. An overview of the broader AI and ML disciplines can be quite helpful in understanding the fundamentals and developing a perspective on the topic.

References

[1] Groce A, Kulesza T, Zhang C, Shamasunder S, Burnett M, Wong WK, Stumpf S, Das S, Shinsel A, Bice F, McIntosh K. You are the only possible oracle: Effective test selection for end users of interactive machine learning systems. IEEE Transactions on Software Engineering. 2013 Dec 12;40(3):307-23.

[2] Tizpaz-Niari S, Kumar A, Tan G, Trivedi A. Fairness-aware configuration of machine learning libraries. InProceedings of the 44th International Conference on Software Engineering 2022 May 21 (pp. 909-920).

[3] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.

[4] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.

[5] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.
[6] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.
[7] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.
[8] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.
[9] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.
[10] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.
[11] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.
[12] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.
[13] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.
[14] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.
[15] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).
[16] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.
[17] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.
[18] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.
[19] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.
[20] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.
[21] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.
[22] Borandag E. Software fault prediction using an RNN-based deep learning approach and ensemble machine learning techniques. Applied Sciences. 2023 Jan 27;13(3):1639.

[23] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.
[24] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.
[25] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.
[26] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.
[27] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.
[28] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.
[29] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.
[30] Khalid A, Badshah G, Ayub N, Shiraz M, Ghouse M. Software defect prediction analysis using machine learning techniques. Sustainability. 2023 Mar 21;15(6):5517.
[31] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.
[32] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.
[33] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.
[34] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.
[35] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.
[36] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[37] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.
[38] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.

[39] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.

[40] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.

[41] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 3: Artificial Intelligence-Driven Test Case Generation in Software Development

Partha Mohapatra

AT&T Corporation

1. Introduction

Modern societies are heavily dependent on software systems of ever-increasing scale and complexity [1-2]. The testing of these software systems is an essential activity during the software development life cycle [3-5]. As practitioners are faced with shrinking deadlines and budgets, a great deal of research in the area of Computer Science has focused on the automation of several phases of the software testing life cycle [6-8]. Such automation aims to reduce the effort required, without adversely affecting the software quality. One of the most critical activities of software testing is the creation of test cases and test suites, the lists of individual tests that will be used during test execution. These tests are typically written after the software developers have finished the implementation and verify the several requirements that a given software system must satisfy. The list of requirements varies depending on the software being tested, the testing strategy used and the aim of the process, for example, functional testing, load testing or security testing. Writing test cases is a repetitive and usually time-consuming task; therefore, addressing the automation of this task has several benefits, including reduction in development effort, enabling developers to focus on other tasks and shortening the time to market.

The automation of test case generation has attracted considerable attention in recent years, especially focusing on reducing human intervention [7,9-10]. A considerable amount of literature has been produced regarding the generation of test cases for functional or load testing. This work focuses on the automation of the creation of test cases for functional testing. When writing test cases, a sequence of inputs that will be used to explore the functionality of the software must be considered; therefore, generating realistic sequences of inputs that will exercise the software is crucial to the creation of useful test cases.

Model-based testing techniques are particularly relevant in this area: a model of either the input domain or the software under test is analyzed to create test case input data satisfying certain pre-established criteria [1,11-14]. Search-based software testing uses particular search heuristics in combination with a suitable fitness function, such as a representation of the adequacy of the current test suite to the testing criterion, to guide the test case generation.

2. Overview of Test Case Generation

Test case generation involves selecting software input values or execution paths for testing and determining the expected test results. Key requirements for test case generation are: it should be both effective and efficient, maximizing testing effectiveness while minimizing working time.

Test cases can be generated manually or automatically. Manual testing can yield good test cases but is inefficient, time consuming, and expensive. Test case generation techniques strive to facilitate this time-consuming process by automatically deriving as many test cases as possible. Due to the extensive use of test cases in checking program correctness, test suites should be continuously refined and improved. Therefore, test case generation techniques should enable the automatic propagation of legacy test cases to testing work for new releases.

3. Machine Learning in Test Data Synthesis

Test data preparation prior to the execution of tests introduces delays in the overall software testing process, which motivates the need for artificial intelligence (AI) assisted test data generation [13,15-17]. Machine learning concepts can be employed to automate test data synthesis by providing insight into existing software functions. One such technique for automatically synthesizing unit tests is forensic testing, which utilizes existing high-quality tests for training a learning model to generate new tests for functions that are not fully covered [18-20].

In forensic test synthesis, a learning model captures general features of the software. Initial executions of the trained model identify more functions, and the newly synthesized tests are then manually assessed by testers [19,21-22]. Mapping functions to existing high-quality tests is non-trivial; while unit test methods can be obtained by searching by the method name, covering tests can be discovered using test coverage tools. The model is trained with all the covering tests for the corresponding function,

allowing the synthesis of new tests that either improve overall coverage or increase diversity in the existing high-quality tests.

3.1. Fundamentals of Machine Learning

Machine learning is an advanced area of artificial intelligence that allows computers to learn from new data and experiences. It differs from classical programming because knowledge is learned and accurately produced by the computer, without any explicit human instruction.

In classification, a dataset consists of a set of input variables, represented by vector x, and a single output variable, represented by y. The goal is to find a function f that can map an input vector x to its output value y. To achieve this, a training set consisting of input–output pairs <x, y> is used to learn the function f. An unseen feature vector is then classified by examining the output of the function, f(x*). Most learning algorithms have a control for model complexity, which is the ability of the model to fit the data.

3.2. Data Synthesis Techniques

Data synthesis is the process of transforming data to support testing activities along a key axis: condition and fact coverage maximization. Condition Coverage Maximization is the process of providing test executions that maximize the number of fulfilled conditions in the story execution. Fact Coverage Maximization is the process of providing test executions that maximize the number of facts logged about the story execution.

The initial data for an STS consists of the story itself, i.e., the list of conditions and facts related to that story. Also, historical data about other protections is considered as input data. The key itself is the creation of new tests that fulfill a condition or generate a new fact in the system—this condition or fact is referred to as the target.

3.3. Challenges in Test Data Generation

The effectiveness of test data generation manifests in several distinct forms, hinging on the nature of the inputs and scenarios produced [11,23-25]. Common distinctions include random, constrained random, model-based or scenario-based, and adaptive or search-based test data generation [26-28]. The capability to generate large volumes of random data serves fundamental testing needs; however, the added value often resides in constrained random, model-based, or search-adaptive generation. The degree of coverage—whether code, branch, or input coverage—often serves as a benchmark for

efficacy. The creation time impacts the overall developer experience; the sheer volume of test data influences rollout time, while the quality affects reliability.

Test data generation confronts numerous inherent challenges. Generating test data for all possible input combinations—possibly infinite in number—remains impractical, necessitating careful selection to balance exhaustiveness against resource constraints. Creating tests that expose elusive errors, often hidden in seldom-executed control paths, represents a complex undertaking. Furthermore, test data must be valid, reflecting realistic operational conditions and satisfying the input constraints of the code under test. Achieving comprehensive ground truth coverage likewise proves difficult, as corroborating expected outcomes for each test case demands meticulous attention.

4. Model-Based Test Case Design

Model-based testing requires a detailed model of the function under test. Test cases can then be generated automatically from the model [29-32]. Naturally, the more detailed the model, the more test cases can be derived. The model can now be employed to generate different levels of testing. It can be used to generate black-box test cases that focus only on the external behavior of the system. More concrete test cases for individual methods of the application can also be generated. These test cases will serve as drivers for the individual components of the system and may be treated as integration tests. When tests are defined with a finer granularity, the model can be used to generate white-box test data that will guide the execution of the application along a particular path in the system.

Despite the fact that a model is at the basis of all phases of software development, in practice, models rarely exist. One of the main reasons for this reluctance to produce detailed models of the application behavior is the sheer complexity of the software. Even a relatively small piece of code could be quite complex to describe. This lack of models can be overcome by generating the models automatically.

4.1. Principles of Model-Based Testing

Model-based testing applies models for requirements specifications, design specification and executable systems as its basis. In systems engineering model-based testing is used to specify and model the test object and to specify and model the tests. During system development, models are used not only for a better understanding of the system but also for assessing its quality [31,33-35]. The quality assessment can be conducted on requirement level, on design level, and on the code itself.

The requirements must not only be tested thoroughly during acceptance phases. Candidate tests should be defined as early as possible during requirement elicitation to derive executable tests automatically, assess the quality of the requirements, and inspect the stated requirements for ambiguities and contradictions. Already during requirement creation, automated quality assessments can support the requirements engineer by identifying inconsistent or incomplete requirements. Moreover, it is possible to determine the quality of the tests, for example the functional coverage, at testing time. The latter can also be applied, when the relevant requirements change and no or too few additional tests have been created during the requirement change.

4.2. Types of Models Used

Generative AI models trained on vast datasets of programming languages are capable of producing country-specific source codes on demand. Prominent commercial options include ChatGPT from OpenAI, GitHub Copilot from Microsoft, and the Amazon CodeWhisperer. Each system supports a wide array of programming languages and establishes connections to Integrated Development Environments (IDEs). They assist users in developing new code, modifying existing code, or generating test cases. Although the underlying generative AI technology is consistent across these offerings, test case generation—now commonplace in the IDE—has distinct requirements, uses and benefits. As a result, test case generation can be seen as a separate category deserving of a dedicated analytical framework. An analysis of these tools is provided in the subsequent sub-section.*Please see the Disclaimer at the end of this article.

Generative AI test case solutions can be primarily classified according to the type of model employed: Large Language Models (LLMs) or Autoregressive models. LLMs are built using a transformer architecture, which enables the models to focus attention selectively on relevant portions of the input encoding [36-38]. In the context of test case generation, the ability to attend to specific inputs or code sections allows the unit test to be tailored to those aspects. In large LLMs, the hybrid transformer plus encoder-decoder architecture also facilitates extraction of important tokens from the input, such as function names. In contrast, Autoregressive transformers employ transformer architectures in the decoder exclusively and may utilize additional techniques like causal self-attention and fewer layers to expedite encoding. With these approaches, while there is no direct 'attention' to specific segments of the input, relevance can be infused into the prompt or initial conditions. A comprehensive overview of LLMs and Autoregressive transformers for code completion can be found in the literature.

4.3. Application of Models in Test Case Generation

Automatic test case generation reduces the time and effort required for testing. Some models guide test case generation toward coverage of specific structural code properties in the final set of test cases [1,39-41]. This can be, for example, to generate test cases that all terminate in an exception state or that cover all data definitions in the test sequence or even to generate test cases that cover targets in a non-reducible path of the tested system. The test paths can be represented or described in a model to examine specific properties using a temporal logic parser. Temporal logic queries in computer science, such as Computation Tree Logic (CTL) allows properties on paths to check the satisfiability on them. Light-weight Behavior Notation (LBN) provides a user-friendly textual syntax for representing behavior in the given sequence. LBN's context-free grammar allows logical properties to be specified in a Behavior property specification language (BPSL). Queries can be applied on the model using model checking tools like mCRL2, which accepts BPSL queries and returns all the behavior paths that satisfy the property.

5. AI for Exploratory Testing

AI can perform exploratory testing through recurrent networks, automating exploratory test in human-centered tasks. In the current paradigm, exploratory testing is a manual process, due to its strength in identifying defects that would be typically missed by any automated test strategy. The manual execution of exploratory tests can be modeled as a sequential activity, where a tester explores the software under testing, observes the behavior of the system, learns from the observed behavior, designs test conditions for the subsequent iterations of testing, executes the test cases, and repeats the whole activity.

Such human approach can be imitated by a Recurrent Neural Network. Utilizing an RNN for exploratory testing involves providing a sequence of activities already executed to the network, which learns the historical patterns to generate new activities that have a very high probability to be explored sooner or later, when a human tester guides the process of testing a software system.

5.1. Role of AI in Exploratory Testing

Exploratory testing defies automation. It thrives on human curiosity, experience, and imagination. It allows testers to explore beyond pre-programmed test scripts, navigating the application in unforeseeable ways. The virtual world presents a natural wine tastings analogy, where humans are connoisseurs tasting a multitude of wines from grapes of

various quality. AI cannot (yet) replace this. Yet, artificial intelligence, in the form of the new and vastly growing exploratory testing bot, can serve as a valuable assistant in this process. It automatically traverses deep into the application, delivering automated yet non-scripted test coverage. Since the testing bot does not sleep and can be set to run continuously, it frequently updates its test coverage, presenting human testers the new changes for human evaluation. Finally, the testing bot can advance much further than a human explorative tester, operating at scale, and browsing millions of possible paths.

The first step in exploratory testing is gaining a broad foundation of understanding: what features does the application support, where do I find interesting data, what are some edge cases, and what mistakes could be useful to test? The researcher "googlathon" tool helps uncover a wide variety of examples, mistakes, or edge cases for a category of input. Natural language can be input; the tool returns a ranked list of common mistakes and examples, organized and color-coded. The next step involves using the information to quickly test an application or API. The tester window leverages the same foundational research, now with user-selectable inputs, to construct and send an API call. The raw JSON response is returned alongside an English explanation, elegant JavaScript code to rescue response or request values, and browser alerts to present the incoming data. Sample code snippets also help verify the success or failure of a request. Finally, the creation of a bot capable of automated exploratory testing of an application based on traffic and responses creates comprehensive coverage over a variety of conditions, evaluate HTML and JSON responses, gather data during exploration, and record a detailed summary.

5.2. Techniques for AI-Driven Exploration

This section discusses four main AI techniques that can be used for the purpose of exploration in the interplay of software testing and AI: evolutionary algorithms, neural networks, reinforcement learning, and probabilistic models.

Evolutionary algorithms mimic the process of natural evolution. They iteratively select individuals with high fitness and variation, recombining and mutating their features to form new populations. The process continues until a termination criterion is met. During test, this approach uses a fitness function to score test cases and rewards diversity among test cases in each generation. Neural networks emulate brain neurons with interconnected dumb cells. Through random initialization and iterative parametrization by gradients, they become specialists in various tasks. For example, to direct test generation, a neural network can encode the system–test interplay so that it becomes a specialist in the «smart most-explorative test selector» task. Reinforcement learning involves an agent learning to achieve goals optimally through trial interactions with the environment. During test generation, the agent explores possible scenarios, supported by

an objective function that provides rewards when states of interest are reached. Probabilistic models estimate the probability distribution of data sources. Their role in testing is to learn the quantitative probability of an action or state, thereby guiding test-case generation probabilities of specific elements.

5.3. Tools and Frameworks for AI Testing

Comprehensive tests ensure the reliability and security of intelligent systems, but the potential number of test-cases grows exponentially with complexity of the modules. Do-it-yourself (DIY) test-case generators mitigate cost of testing, but synthesis, minimization and hiding usually require separate tools. An integrated framework controls parameters and cost, and covers a broad range with a single call of the generators.

Open-Source Software (OSS) is popular due to its cost, quality and reliability. But, OSS is also vulnerable to malware. Software Assurance (SwA) protects against malware and external attacks by comprehensive tests throughout the life cycle, including the validation stages. A Dish of the Day (DOD) generator produces effective tests for any component and at any stage. It presents a long list to inspect after the hash code. A Combination of Ingredients (COI) generator synthesizes variants and minimizes the number of test cases to suit the budget. A Cook's Black-Box (CBB) generator conceals the test cases for hidden and closed source software.

6. Integration of AI in Testing Workflows

A crucial priority in the advancement of artificial intelligence (AI) for software testing is the integration of AI techniques and algorithms into the entire workflow and toolchain for automated software testing. Such integrations are essential to realize the benefits that AI approaches can deliver in reducing the costs and improving the quality of software testing.

Effective workflows have been developed that combine the powerful capabilities of deep reinforcement learning (DRL), evolutionary-fuzzing-based test generation, and service-hooking-based test execution to produce an efficient test oracle capable of reducing manual checking effort. Moreover, a cloud-based GUI testing environment utilizing a heterogenous approach has been proposed, integrating exploration-based, bias-based, and machine learning (ML)-based testing methods to achieve high code coverage and fault detection rates. Additional workflows have been applied to the challenge of exploratory testing, demonstrating how the expertise of a human tester can be augmented

with AI methods. Finally, a new approach implements a coactive testing framework where the human tester and an AI agent collaborate.

6.1. Continuous Integration and Testing

CI is a software engineering practice in which members of a team integrate their work frequently, resulting in multiple integrations per day. Each integration is verified by an automated build that allows teams to detect problems early.

Test cases represent a test step, or a logical set of test steps, for validating software system functionality.

Continuous testing applies automated tests at every stage of the software delivery pipeline. Often, those tests are executed in an isolated environment without impacted dependencies and mocks. Integration of CI and CT allows the creation of core test cases and use cases based on user stories and ticket details, which facilitates developers' work in completing automated scenarios.

Collaborative workflow optimization automatically generates test cases and test scenarios, increases their reusability and integrates the process of creation, rescheduling, executing, tracking and controlling process of test cases and test scenarios. AI techniques for exploratory testing allow inclusion of test characteristics and API model, allowing a series of optimization functions to be used.

6.2. Collaborative Testing Environments

Quality assurance and software testing are crucial components of software development processes. However, these tasks are often considered time-consuming and dull, especially at scale. To alleviate these issues, artificial intelligence can be applied in software testing.

Inspired by the advantages of collaborative working environments, the concept of collaborative testing environments aims at supporting testers, developers, and other stakeholders during exploratory testing activities. Exploratory testing considers the experience and investigation skills of the tester. Similarly, the environment explores the system under test. It proposes test actions that the tester can consider and perform. The environment is inspired by automated exploratory testing using artificial intelligence techniques such as reinforcement learning or heuristics-based approaches. These investigations leverage the experience of the tester and propose interesting, unexplored parts of the system for further testing. The collaborative testing environment concept supports both the tester and the automated exploratory tester and can be adapted to suit

the team. A shared driver manages the automated browser instances for testers and the artificial intelligence, preventing any possible conflicts. Additionally, a socket communication enables interaction between the automated exploratory tester and the integrated chat functionality in the environment. This support allows testers to exchange their experiences and enrich the environment with additional input about the system. The environment supports a broad spectrum of exploratory testing activities and testing stages in the development lifecycle.

7. Case Studies

Several studies have investigated how new software functionalities with large and complex input spaces are tested through software development. Test case generation with the aid of AI techniques is a prominent research area.

AI-based approaches usually train a model with a dataset of requirements and the corresponding test documents, and then generate complete test documents or numerous test cases. For example, Li coded the test inputs in binary format and fed them into the LSTM code generator for Android Intents. In contrast, Liu trained a Seq2seq model using input/output pairs extracted manually from the JUnit test class ShareFile.

7.1. Successful Implementations of AI in Testing

Artificial-intelligence-driven test case generation is a relatively new approach to software testing and to date has exhibited support for nine languages and three requirement types. Many projects are especially challenging because, in addition to having a large number of functions and data types, they also display many requirements that traditional models cannot capture. The literature demonstrates the applicability of this approach in three domains: telecommunications, multi-agent, and medical emergency applications. Results indicate the number of generated test cases consistently exceeded those stated in the requirements, and the success rate of the generated test cases likewise surpassed the success rate of tests conducted by experienced human testers. Applications are therefore amenable to testing via model-driven test case generation. The use of biologically inspired techniques, such as heuristics, in software engineering enables the development of methods and systems for testing or optimizing any software application.

Modules can be automatically generated by establishing the Boolean connection of all predicates together with expected results—TRUE or FALSE—represented by a bit pattern. The application of metaheuristics facilitates development in many areas, including software testing. In artificial intelligence, agents are software entities whose

perception and action spaces allow them to act, perceive, and be aware of their environment. Nevertheless, requirements are not always completely specified; requirements problems may be solved incorrectly or not solve the problem at all. Software testing ensures system functionality aligns with specifications. A telescope continuously measures the azimuth and elevation of the telescope's guide star relative to subreflector positions for its guiding PID loop. Testing of such an interactive system using traditional techniques is time-consuming and prone to error. Software testing employs not only metaheuristics to generate test cases but also considers the environment in which the software functions. Testing of similar network systems using metaheuristics has been carried out in the telecommunications domain.

7.2. Lessons Learned from AI-Driven Testing

Section discusses the experiences and insights gained from employing AI methods in test specification and test case generation. Software systems are enduring complex digital transformations, and these changes have profound impacts on quality assurance activities. One approach to test automation tackles the problem at the specification level by producing test cases from models automatically. New research in these areas includes applying AI methods for test specification and test case generation.

In response to the global COVID-19 pandemic, the U.S. government implemented large-scale web-based systems for benefits distribution and management. For example, during the initial outbreak, the US Internal Revenue Service launched a web-based system to distribute approximately $270 billion in Economic Impact Payments (EIPs) to eligible recipients. Likewise, the New York State Department of Labor introduced its Pandemic Unemployment Assistance (PUA) system to deliver over $63 billion in unemployment benefits to more than 3.2 million claimants. The limited time available for deployment, combined with the massive scale and intense public scrutiny, resulted in new systems beset by bugs. Traditional manual software testing approaches were insufficiently scaled to adequately address the volume, velocity, and demand for effective testing. Consequently, the use of Artificial Intelligence (AI) for test case design and selection received increased focus and served as a catalyst for rapid and substantial innovation.

8. Future Trends in AI and Testing

The use of automation in test creation has just scratched the surface. Automation is entering a new phase, in which it builds on its success of making test execution more efficient, heading towards efficiency also in test design. This evolution accelerates with the help of Generative AI, that is, Large Language Models, like ChatGPT. Introducing the first steps that leverage Generative AI, companies benefit from notable time savings

in testing activities. The most significant advantages are, therefore, time savings on test creation, but also better test coverage for important scenarios.

Generative AI can serve multiple functions within the testing domain. First, it can simplify the transformation of user stories and high-level test scenarios into detailed validation steps, making testing more accessible to testers without a coding background. Second, by suggesting improvements to existing test cases, Generative AI helps maximize the effectiveness of already generated tests. Finally, it can produce executable code or scripts for automated tests, significantly accelerating the overall testing process.

8.1. Emerging Technologies in Software Testing

Deep learning methods such as Neural Language Models dramatically change the way test cases are generated in software testing. Test Case Generation with Neural language models allow testers to achieve better code coverage and reveal more bugs early in the software development process, which help create more reliable software and increase users confidence.

Deep learning (DL) methods such as Neural Language Models (NLMs) also considerably change the way test cases are generated. Research at the School of Computer Science and Engineering, Nanyang Technological University, Singapore, has verified that the automatic generation of test cases based on NLMs achieves better code coverage and reveals more bugs early in the software development process. That helps create more reliable software and increases users' confidence.

8.2. Predictions for AI in Test Automation

The prospect that increasingly sophisticated AI technology will be able to generate software test cases is compelling. Indeed, much recent research has databased software development, examining large repositories of projects and the issues that have arisen in order to gain a greater understanding of complex software systems—and then using the generalized knowledge obtained to generate new test cases. A constantly growing interest group from the fields of Machine Learning and Theoretical Computer Science are beginning to apply Machine Learning to software testing.

DeepTest is an example of AI-based techniques for automatically testing self-driving cars operating in a simulated environment. After training a Deep Neural Network (DNN) to detect the steering angle within a self-driving car, the approach automatically generates test cases for the DNN by creating synthetic test images. Along comparable lines, Skinner is a tool that automatically generates unit tests for an existing Java method. By mining source code from popular open-source software projects, Skinner extracts

and learns code parameter–return value mappings to create method-level unit tests. An early approach to describing an Application Programming Interface's expected behavior using Machine Learning was the Randoop testing framework. Randoop combines sequences of existing methods to form new tests. Once these tests encounter new program states, Randoop automatically asserts the observed new conditions.

9. Ethical Considerations in AI Testing

Ethical considerations extend into AI testing, with Principles of AI Ethics recommended for incorporation into testing plans and frameworks. Bias of AI can thus be reviewed. Following Ethical AI Principles will also help to realize Responsible AI.

For producing intelligent apps with highest standards of quality, Testing Deployment Pipeline with AI should be set up. This would have AI-powered AI models using ethical AI principles, and such models would be handling all testing related aspects of AI-powered applications. Recent Examples include the DeepMind AI ethics demonstrator built upon the Equilibrium AI framework, and Microsoft's responsible AI dashboard

9.1. Bias and Fairness in AI Models

AI-Driven Test Case Generation offers many advantages for automated testing; however, certain risks warrant consideration. Incorporating a test case generation mechanism driven by external test results can potentially jeopardize the software quality assurance process, introducing Algorithmic Bias—a phenomenon where AI-generated test cases exhibit an unfairness or unfavourable treatment of one or more classes within the Software Under Test (SUT) prediction space. Sources of bias include imbalanced data, skewed test results, feature engineering oversights, selection of algorithms, and finally, the model evaluation process. It is vital to assess the fairness of AI-generated test cases by analysing the quality of SUT test results and the subsequent generated test cases.

In fairness-aware AI models, a Fairness Evaluation stage is added to the AI lifecycle. This stage evaluates the quality of the machine or deep learning models based on the presence or absence of bias and discrimination. Bias and fairness metrics for both classification and regression problems guide this evaluation process. If the model is found to be biased or unfair, a Bias Mitigation stage follows. This stage aims to alleviate bias by either preprocessing the input data, modifying model training and prediction algorithms, or post-processing the prediction results.

9.2. Transparency and Accountability

Transparency is an essential feature for accountability and trust in any model. It requires understandable, nontoxic outputs as well as simple visual cues and the symbols that identify a model and reveal its underlying characteristics. Simplified model details or badges also communicate important context to the end user.

Model reporting includes any information that clarifies the characteristics of a particular LLM under a common structure. This comprises technical specs, usage patterns, high-level documentation and metadata and model details expected by the business, local, state or federal government. For example, evaluation results enable the assessment of utility and risks. The format of model cards can vary depending on the type of model and target audience.

10. Conclusion

Ensuring the quality of evolving software is among the most complex and expensive tasks. Thorough software tests are essential yet notoriously time-consuming to produce. The combination of AI with the data contained in the Version Control System and its integrated issue tracking systems can largely automate the creation of tests. AI models are capable of analyzing heuristics in the form of previous changes to the source code and generating unit tests accordingly. Although these automatically generated tests require review and completion, their scrutiny is considerably faster than the creation of new tests from scratch and still yields high quality. In the long term, AI-driven test case generation will have the potential to save much time in software testing and contribute to better software quality.

References

[1] McMinn P. Search-based software testing: Past, present and future. In2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops 2011 Mar 21 (pp. 153-163). IEEE.

[2] Davis CG, Vick CR. The software development system. IEEE Transactions on Software Engineering. 2006 Sep 18(1):69-84.

[3] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.

[4] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.

[5] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.
[6] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.
[7] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.
[8] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.
[9] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.
[10] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.
[11] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.
[12] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.
[13] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.
[14] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.
[15] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.
[16] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.
[17] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).
[18] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.
[19] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.
[20] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.
[21] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.
[22] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.

[23] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.

[24] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.

[25] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.

[26] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.

[27] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.

[28] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.

[29] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.

[30] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.

[31] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.

[32] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.

[33] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.

[34] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.

[35] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.

[36] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.

[37] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.

[38] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.

[39] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.
[40] Juristo N, Moreno AM, Strigel W. Guest editors' introduction: Software testing practices in industry. IEEE software. 2006 Jul 17;23(4):19-21.
[41] Wiklund K, Eldh S, Sundmark D, Lundqvist K. Impediments for software test automation: A systematic literature review. Software Testing, Verification and Reliability. 2017 Dec;27(8):e1639.

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 4: Predictive Defect Analysis in Software Engineering: Approaches Using Artificial Intelligence and Machine Learning

Partha Mohapatra

AT&T Corporation

1. Introduction

Newton's second law states: For a body of constant mass, the resultant force acting on it produces an average acceleration in the direction of the force which is directly proportional to the magnitude of the force and inversely proportional to the mass of the body [1-3]. The basic idea that lies in the predictive defect analysis scheme can be summarized as follows. The reduction of acceleration from its undefected reference value to zero due to the resulting unbalance force of the rotorâ€™s masses weighted with the operating speed and the path length provides indication of the amount of damage (or defect severity). Furthermore, the reduction of acceleration from this maximum value at the undefected reference state to low level during the accumulation of the damage provides an indication of the threshold level [2,4,5]. In the vibration monitoring of rotating machinery, faults are often connected to changes in the vibration pattern of acceleration. An acceleration-based predictor can be defined as the difference between the vibration acceleration measurement at the reference state (zero damage) and the actual response at any other pathological state. Experimental results verified that the acceleration- and the conventional displacement-based predictive defect analysis techniques provide alike information on the evolution and accumulation of the damage.

2. Historical Defect Pattern Mining

Defect data repositories offer a rich source for data-mining activities designed to uncover historical information on defect patterns. This process examines repositories housing information on the phases or releases traceable to defect patterns, enabling the

identification of critical assumptions in prior repositories, extraction of statistical patterns in historic defect data, and evaluation of predictive capabilities in defect statistics derived from legacy datasets [6-8]. The historical information discerned can, for example, pinpoint defects exhibiting similar characteristics—as defined by working assumptions—that tend to reoccur. For a historical defect pattern repository to serve effectively, it must encapsulate information about defect recurrences (do they indeed recur?), the defect timings (when do defects recur?), and the defect repositories involved (in which repository does the recurrence occur?).

Mining historical defect patterns involves extracting significant patterns from repositories that document defects observed in various products. These repositories are collected across phases and releases, each area offering unique insights into the recurring nature of defects. In broad terms, effective repository formation necessitates addressing three pivotal questions: which defect types recur, at what junctures do they recur, and in which repository or product does this recurrence manifest? Addressing these queries contributes to forming a holistic framework for historical defect pattern mining.

2.1. Overview of Defect Patterns

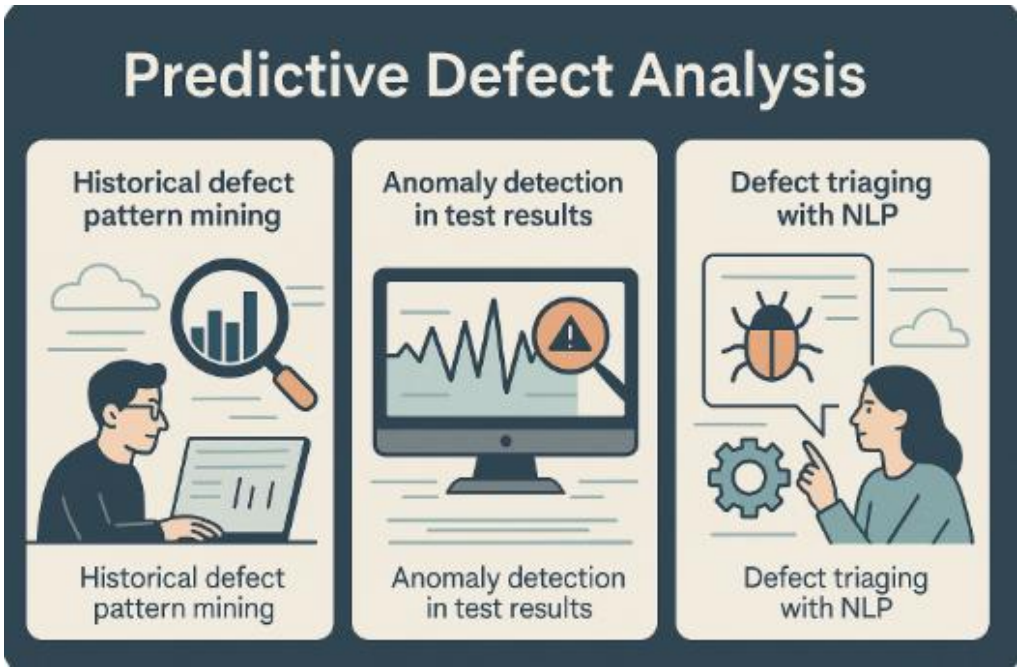
Many defects result from common mistakes that programmers repeatedly make. For instance, new object allocations followed by calls to specific methods (such as prepare) are often overlooked [9,10]. To detect such errors early, relations can be documented as \("Straw Man"\) patterns for later formalization, and standard-types generalized by signatures to focus on particular aspects: \(\textsf{newObjectAllocation}\). Subsequently, the program's call graph is checked to ensure \(\textsf{prepareMethodCall}\) is involved. Despite providing early feedback, this approach requires domain experts to manually specify these relations and patterns.

Higher-order pattern generalization is also possible. Standard types are not merely signatures, for which the Suffix Array can spot duplication, but also vectors of signatures, for which the construction of a Scale Array reveals duplication of whole vectors. Finally, partial patterns are useful when a sequence of calls must be executed in an exact order but calls that are missing from the sequence are still allowed elsewhere. Partial patterns, as described by Thomas., can lead to more expressive results and pinpoint applied numbers for a specific rule.

2.2. Data Collection Methods

Various methods can be applied for data collection in defect prediction, such as analyzing the modification history or mining e-mail archives, which preserve knowledge

about reported defects. In a few organizations, databases for customer support already exist and can be used in this context. Several approaches have been developed that exploit information contained in software repositories like CVS, git, or SVN. Such data is similarly available for nearly all software projects. For each delta, numerous characteristics, including comments or author information, are readily accessible and can be evaluated. A modification to the source code typically causes defects in the modified module. The authors of that modification are well suited to correct those defects efficiently and are, therefore, suitable defect reporters. Thus, it is expedient to evaluate the frequency and the amount of modifications for each source code file.



2.3. Pattern Recognition Techniques

Patterns can be defined as a set of intelligible relationships, any of which can be recognized as similar to other, complex relational sets [11-13]. Pattern Recognition is a branch of machine learning that focuses on the recognition and classification of data based on either a priori knowledge or information extracted during the learning process. The classic pattern recognition problem involves assigning a label to an object based on a set of measurements, with objects generally being classified into one of several known classes. This entails the detection and exploitation of regularities and correlations in the data, frequently leveraging methods such as machine learning or data mining.

Up until the 1990s, pattern recognition was treated as a subfield of both artificial intelligence and statistics, employing techniques and theories from both domains.

Subsequently, it became recognized as an independent academic discipline. Although pattern recognition shares certain characteristics with other areas—artificial intelligence being considered its rational reasoning part, and data mining addressing the problem of discovering patterns—it maintains distinctive approaches. Support vector machines, a methodology developed in the pattern recognition community at the same time as neural networks, are now employed in data mining as well.

2.4. Case Studies in Pattern Mining

A variety of mining techniques have been applied to fault prediction. G. M. Kapetanakis (Overall lookahead for software fault prediction) used association to detect patterns in the changelog. P. V. Krishna and R. Zafar (Mining software fault patterns using frequent itemsets) showed that fault patterns can be extracted from the most frequent itemsets in the change history of fault-fixing edits. M. R. B. De Medeiros (Mining bug repositories for associating bug reports with debugging changes) extracted the Debugging Pattern from logs of a fault database.

L. Yu (Stable cross project defect prediction) demonstrated that particular types of faults tend to re-occur frequently. Finally, A. Hindle (What Would Your Code Look Like if It Was Trendy) introduced a fault trend called the Decay Bug.

3. Anomaly Detection in Test Results

Anomaly detection is a category of algorithms whose goal is to determine which regions of the space are considered anomalous, i.e., not conforming to the concept of normality [2,14-17]. Distinguishing between anomalous and normal observations poses an inherent problem regarding supervisory information; therefore, only supervised, semi-supervised, or unsupervised techniques are available. Supervised techniques require incorrectly labeled (anomalous) observations during training, which presents some drawbacks: anomalies are often scarce, difficult to anticipate, label, or even unknown during training. Semi-supervised techniques are trained with only correctly labeled observations, whereas unsupervised techniques assume that anomalies are statistically unlikely and apply this model directly to the sample space.

The main methods for semi-supervised anomaly detection have evolved from traditional machine learning algorithms, such as One-Class Support Vector Machines (OCSVMs), nearest neighbors, or Bayesian networks [9,18-21]. Those methods have also been implemented using Artificial Neural Networks (ANNs). Recently, Deep Learning (DL) techniques, able to extract features from the raw data, have proved promising in the aims of anomaly detection; some examples are variations of Convolutional Neural Networks

(CNNs) for spatial data or Recurrent Neural Networks (RNNs) specialized in temporal sequences. Although it is a non-trivial task due to the nature of anomaly detection, supervision remains a key element in the task of training deep anomaly detection networks: not only labels are normally needed, but they must be precisely determined at the observation level.

3.1. Definition of Anomalies

What should be detected as an anomaly is neither well-defined nor universal. The criteria vary with use cases and are usually subjective [22,23]. These inconsistencies have led to many expressions, such as outliers, novelty, noise, exceptions, surprise, or inconsistencies. The capacity to detect specific types and subtypes of anomalies varies with the categorization used. Twelve major categories of anomalies can be distinguished, each containing subcategories. A specific use case may require detecting only a subset of these categories. Credit card fraud detection offers a well-documented example. ‘‘Other fraud’’ concerns transactions made with a legitimate credit card and stolen personal credentials (such as a bank code or a social security number). ‘‘Lost credit card’’ relates to the use of a blank card that is lost and later used by a thief to make electronic purchases [24-26].

3.2. Statistical Methods for Anomaly Detection

Statistical methods for anomaly detection describe a broad field of data mining that identifies deviations from normative behavior and draws special attention to the detected change [27,28]. Like the rule-based methods, these approaches can be applied across many fields. Such statistical approaches are based on the premise that the data of interest is mean-stationary. That is, historical data reflects normal behavior, and no underlying concept drift is present. The Quantile method represents the distribution of the data with quantiles, and when applied to streaming data it tracks whether new data points fall outside of the modeled distribution. A related Quantile method, HOT SAX, utilizes the symbolic representation of the time series to avoid transformation of historical data during streaming analysis. The Donut approach applies an unsupervised deep anomaly detection model for water level sensors of the Hong Kong Mass Transit Railway. The model achieves good precision and recalls without utilizing labels of known events.

3.3. Machine Learning Approaches

Machine learning (ML) is the use of algorithms that learn from experience. There are several machine learning paradigms. Among them, supervised and unsupervised

learning are the most common. During supervised learning, the algorithm learns from labeled data. Given a set of attributes as the input, it tries to predict the label of the data. Supervised learning algorithms can be used for classification or regression. During classification, the algorithm tries to find a label categorizing the data—for example, the label of a movie review given its text could be positive or negative. During regression, the algorithm tries to predict a numerical value as the label, such as the expected life of a machine given recent condition data.

During unsupervised learning, the data has no label; instead, the algorithm tries to understand the structure of the data. One of the most common tasks during unsupervised learning is clustering, which finds similar clusters of data. For example, in the case of the movie review, unsupervised learning algorithms could group similar movie reviews, perhaps grouping horror movies distinct from action films.

3.4. Metrics for Anomaly Detection Evaluation

Evaluation metrics for Anomaly Detection are discussed. Good metrics should be independent of threshold selection and robust to skewed class distributions for meaningful comparison of different approaches. Metrics that directly measure the similarity between the actual and the detected anomalies provide important insight into the detection quality. A row in the results table corresponds to a single anomaly and columns are evaluation metrics. Each metric views the anomaly detection result using an evaluation perspective which is the foundation of the evaluation metric.

The Point-Adjusted metric modifies the label of each detected anomaly point to consider whether any element in the respective anomaly segment was detected correctly. The idea is to reward detection models that flag some part of an anomalous segment to measure the performance of the detection algorithms from an application perspective. The resulting metric assigns a label of 1 to every data-point in an anomalous segment if the model flagged some data-point in that segment correctly (whether at the start or not). However, it incorrectly assigns a label of 1 to the entire segment even if the model flagged only a middle point of the segment. This leads to over-optimistic results if the detected point lies in the middle.

4. Defect Triaging with Natural Language Processing (NLP)

Text classification techniques can facilitate defect triaging and thus reduce the human effort spent in defect management. Sentiment analysis offers useful hints for prioritizing defects [19,29-31]. Methods that classify defect reports into categories such as English, non-English, duplicated, invalid, or delivered-as-designed can reduce the effort

developers spend on triaging. Although defect triaging can benefit from the rich information encoded in defect reports, approaches that utilize this information for triaging are still in a preliminary stage and face multiple challenges.

Predictive defect analysis is important for software quality assurance [32,33]. Testing is expensive and resource-intensive, making it infeasible to test everything all of the time. A mechanism that can prioritize which test cases are run will better allocate the limited testing resources. The research covers mining common defect patterns from historical defect information and detecting anomalies in software test results. Text classification techniques are used to facilitate defect triaging. The choice of testing priorities can be guided by the likelihood of a test case failing, which can be determined by analyzing the results of previous test runs. In this context, anomaly detection techniques may be employed. Historical defect data provides support for mining defect patterns, which is useful for prioritizing mutants during mutation testing.

4.1. Introduction to NLP in Software Testing

With the large amount of text data generated during software testing, it is only natural to explore whether there are ways of analysing it using techniques originating from natural language processing (NLP) [34-36]. The general argument lies in the quest for automation: can NLP methods automate certain aspects of software testing? One aspect that has received particular attention is the area of defect triaging, where NLP methods have been applied in the use case of classifying bug reports according to their severity or priority levels or in an approach supporting the triaging by including an analysis of the sentiment demonstrated in bug reports. Nevertheless, these approaches deal mainly with the textual information of the defects and do not typically consider other sources of information.

By contrast, the approach of predictive defect analysis aims to be predictive and proactive, with an emphasis on mining historical defect patterns. Such patterns provide useful information that can support early defect-prone code and module detection for effective allocation of resources. They also help to predict quality issues for functionalities. Finally, they aid in identifying potential problem areas of code and test cases. The text data arising in this setting are the collected descriptions of actual past defects. Therefore, the focus for this approach lies not only in NLP but also in the general field of anomaly detection.

These two directions—historical defect pattern mining with an anomaly detection focus and NLP for defect triaging—represent different manifestations of predictive defect analysis. Since the former direction is guided by the mining of historical defect patterns

with the latter direction in mind, the current coverage likewise starts with a discussion of the relevant defect-pattern-mining activities.

4.2. Text Classification Techniques

Different text classification approaches have been used to help automate the defect triaging process [37-40]. Duric and Trimm employ hierarchical clustering, while Lau utilize decision trees. Chawla present a focused crawler approach based on classifying bugs, and Cubranic and Murphy integrate KNN, Naive Bayes, and SVM classifiers.

Sentiment analysis techniques have also been investigated for triaging purposes. Mack extract subjective excerpt features and syntactic features from bug reports before applying multi-class Support Vector Machines (SVM). Jabangwe deploy sentiment classification using an ensemble of classifiers comprising Logistic Regression, Naive Bayes, Support Vector Machines, Random Forest, Stochastic Gradient Descent, and Extreme Gradient Boosting. Combining historical defect pattern mining and anomaly detection can not only reduce false positives but also provide context to the predicted anomalies by indicating the types and the root causes of similar defects that occurred in the past. Additionally, triaging can be automated using Natural Language Processing (NLP) techniques.

4.3. Sentiment Analysis for Defect Prioritization

Natural language processing (NLP) techniques can aid other advanced software testing methods, such as defect triaging and prioritization. Prioritization can be accelerated by text classification techniques and sentiment analysis. A drawback of NLP methods is that they typically require significant training before either producing meaningful results or supporting other prioritization techniques.

Sentiment analysis might also be applied to requirements for making a more precise, contextualized prediction of the severity of defects related to a single requirement.

4.4. Challenges in NLP for Defect Triaging

Natural Language Processing presents obstacles from multiple directions in the context of automated defect triaging. Such challenges include noise, unstructured text in issue reports, data quality, and imbalanced class distribution. Noise, often derived from typos, syntactical errors, and inconsistencies in the data, may lead to overfitting prediction models even before the data processing stage. Besides being a time-consuming, tedious, and manual task, issue report preparation is also prone to producing unstructured or

ambiguous texts that are difficult for computers to parse correctly. The broad range of variations in structures, expressed by issue reporters during issue submission, largely differentiates texts. Data quality may vary due to inconsistencies during the issue submission, which, in turn, reduces the quality of classification results. Furthermore, the class distribution of the assigned classes may vary widely due to a few classes being highly populated and the rest with very few samples; these are known as imbalanced classes. This imbalance may misguide the prediction model, resulting in bias toward the highly populated classes, thereby affecting prediction performance.

5. Integration of Techniques

Static code analysis methods can be leveraged for predictive defect analysis. Defect data can be consolidated with relevant metrics—such as coverage, complexity, and code churn—that have demonstrated correlations with defects. By combining defect data with metric data in a predictive data set, it becomes possible to predict defect content at various granularities—such as the product, release, file, and component levels. These predicted defect contents provide insights into the expected number of defects in untested code, thereby facilitating the prioritization of testing and inspection efforts.

Implementation of these predictions in a Microsoft Excel™ spreadsheet, which utilizes values produced by a static analysis tool as input, enables a straightforward approach for leveraging the data. Extensions to this predictive model include the correlation of functional code coverage with defect content and the application of model-generated test plans as inputs within a test path analysis process.

5.1. Combining Pattern Mining and Anomaly Detection

Frequentbugmining applies association rule mining to summarise the patterns of bug reports. The technique tackles bug triaging by identifying the truth label of the reports that have been filed by the bug reporters with a bad accuracy or quality of bug-filing history. Typical patterns of fixed bug reports are mined from the database of historical reports. Anomalous reports whose features deviate from the patterns of fixed bugs are detected as duplicates or non-bugs. Duplicate bug report detection detects earlier duplicate reports that describe the same underlying fault as the new reports. The detection is typically conducted by the component or product in the mapping from new reports to the repository of existing reports. The triaging decision in Bugzie is operated only on non-duplicate reports. Bug piste predicts the operational states of bugs. Covering reports are those fixed bug reports that have been burned for covering the next release. Prediction is performed on the non-covering reports.

Frequentbugmining transforms the bug triaging problems, including bug severity detection, bug reopening prediction, and bug-proneness prediction, into pattern mining problems. Function-level defect prediction is not yet realized, due to the inefficient transformation for frequentbugmining. The inefficiency is caused by the sparseness of the bug-function matrix. The solution integrates frequentbugmining and Pule. The new offensive is based on the observation that bug reports are spatio-temporal covers of the fault states reported by the historical related bug reports in the fault state transition model.

5.2. Leveraging NLP for Enhanced Analysis

Natural language processing (NLP) facilitates the analysis of textual information, be it linguistic or code. The availability of sophisticated output representation techniques further enriches the analytical potential. NLP applications enable insight extraction without incurring the costs associated with large-scale human investigations.

The textual data that issue tickets contains code artifact references, summary descriptions, and elaborate discussions—forms a valuable corpus for software defect analysis. Consider a scenario where individuals external to the development team raise concerns regarding potential issues. The initial descriptions within these tickets allow managers to grasp the problem's nature, anticipate delays, and predict possible deficiencies. An astute examination of the wordings in these tickets not only identifies typical flaws but also anticipates undesirable project outcomes.

6. Tools and Technologies

The development of predictive defect analysis methods requires software engineering data with rich defect information associated with specific software components and a historical perspective that traces the changes across components and tracks how these changes are related to defects. Available databases and related tools include several of the following. These approaches produce historical and defect measurement databases required for predictive defect analysis.

An integration of four components is necessary: (1) a configuration management system that tracks all changes to software components, (2) a defect tracking system that monitors and stores all defects found at various stages of the software development, (3) an access interface that links the information from configuration management and defect tracking systems, and (4) a historical database that stores the collected software history and defects. The links provided by the access interface between defects and the components

changed to fix these defects enable analysts to view the software history and defect information from different perspectives and formally investigate their relationships.

6.1. Software Tools for Defect Analysis

Predictive defect analysis can follow a variety of paths depending upon the software tool employed Focus has been placed on the methods and mechanisms that can produce highly abstract software products Fantasy is easier for the software tool to manage when the design is so abstract and it can then be compared with the implementation in turn A well-supported software tool is usually of commercial origin and can be expensive Incorporation of the project methods and mechanisms will help focus the software tool at the appropriate level Fantasy success often means breaking down the problem into components and concentrating on those abstractions

Object oriented techniques for software tools normally produce a less abstract software product than other methods because valid and recognized implementation mechanisms are not fully supported Games, simulations and fantasy are more difficult to manage because the design and implementation are normally only loosely connected. Management support from all specialisms is vital, along with tools to deal with the interchange of information and ideas in order to gauge the size of real fantasy success even though the conceptual framework is more general.

6.2. Machine Learning Frameworks

The Machine Learning Framework uses defect detection metrics to classify whether a given checkpoint is 'Good' or 'Bad' and can subsequently forecast future commit risks. The study contrasts the Defect-Readiness metric with various established metrics to gauge these frameworks' robustness; comparable results would underscore the metric's applicability and generalization. Data used for training is extracted from the SZZ algorithm's simulation, which generates a labeled training set describing commits up to a certain period in the past. Checkpoints labeled 'Good' serve as positive examples, those labeled 'Bad' as negative, following the class distribution ratio. To prepare the datasets, various commit sizes—ranging from 1 to 10, 10, 50, 100, 500, 1,000, and up to the entire dataset—are selected. The study considers multiple classifiers available in the Scikit-learn library: Logistic Regression, K-Nearest Neighbors, Support Vector Machine, Random Forest, Ada Boost, and Naïve Bayes.

6.3. NLP Libraries and Resources

Several NLP libraries enable text processing in a variety of languages. The MainNLP library1 represents a Java set of NLP tools with pre-trained statistical models such as tokenization, lemmatization, deterministic part-of-speech (POS) and named entity recognition (NER); additionally, the library provides deterministic open information extraction. The NLP4J tool2 offers POS tagging, NER, dependency parsing and deterministic open information extraction for English texts. SpaCy3 supports 7 languages and 72 trained statistical models for POS and NER. Its tokenization approach is rule-based, but it supports different algorithms for sentence splitting that can either be rule-based or learned. Stanza4 provides tokenization, lemmatization, POS tagging, NER and dependency parsing models for 66 different languages and supports training custom models. It also offers biomedical specific NER models. OpenNLP5 implements maximum entropy machine learning models in several languages for tokenization, sentence splitting, POS tagging and NER.

Available NLP resources include PyMystem62 morphological analyzer for the Russian language, which is capable of lemmatization and provides information about part of speech. The BERT language model 7 provides pre-trained masked language models with their contextual embeddings, BERTology probes related models and their performance on various datasets. BERTology also compares multilingual cased and uncased model variants (bert-base-multilingual-cased, bert-base-multilingual-uncased and xlm-roberta-base).

Different resources represent pre-trained BERT-based models that vary either for the Russian language or the software engineering specific domain using the Hugging Face library. rubert-base-cased 8 is originally trained on Russian news. deepPavlov/rubert-base-cased-sentence 9 is a rubert-based model with a Siamese network. cointegrated/rubert-tiny 10 and cointegrated/rubert-small11 are optimized for a small size and fast inference time. uklfru/roberta-russian-base-academic12 is adapted on academic literature. dulny/roberta-russian-sentence-level13 supports tasks on the sentence level. Sberbank-ai/sbert_large14 is the Russian variant of SBERT. БЭРТ 15 (an acronym for the BERT used for Biomedical Text Mining) is an Electroencephalography diagnostics domain specific Russian BERT model—ruELECTRA 16—implemented on the ELECTRA architecture for the Russian language. RuCodeBERT 17 is a code understanding and code documentation generation pre-trained model for the Russian language. CodeReview18 represents the code review dialog BERT-based model. Sophisticats's transformers-opt19 support Russian POS.

7. Case Studies and Applications

Case Studies and Applications

The predictive defect analysis approach has been piloted in a range of software ecosystems to demonstrate its practical utility. Recent industrial case studies of static code-analyses in use for defect prediction and prioritisation provide additional perspective. On one hand, Black Duck Software (2010) demonstrated the use of Product metrics (lines of code, number of modules, complexity, coupling, number of developers, number of change requests, number of code changes and number of bugs in file) to build defect prediction models. On the other, Juliusson used Product, Process, People and Review-Audit metrics. Their data originated from two static-analysis tools.

Industry has been shown to invest in defect-prediction tools, but with limited success. One explanation is that many of the studies have not taken into account the actual information needs and constraints of practitioners. A further study therefore elicited perceptions from practitioners on software defects, predictive models, and updates during the software lifecycle. Next, metrics of a single Eclipse product builds were sourced. These were combined with the practitioner feedback and used to build defect prediction models. The research objective was to identify the Build metrics that are useful for Defect Prediction in Continuous Integration Environments, such as Jenkins.

7.1. Industry Applications

Significant advancements have been achieved in applying predictive maintenance to optimize production operations and minimize unplanned machine downtimes in manufacturing plants; nonetheless, the range of industries deploying these techniques worldwide remains limited. The requirements of industry dictate a heavily supervised approach, frequently supported by chemical engineers on the shop floor and engaged in failure inspection and diagnostics whenever machine breakdowns occur. Feedback from these engineers serves as a practical validation of the implementation. Consequently, results become reliable only when the feedback system is consistently updated by the engineers.

Investigation priorities shift as per industry necessities. For example, a factory in Sriracha, Thailand, producing surfactants for products in the personal care and home care markets, encounters a significant Share of Value loss—amounting to billions of THB per year—during the crystallization operation. Stage three focuses on exploring and monitoring defects and their causes in the crystallization process with the aim of reducing the share of production loss during defect production.

7.2. Success Stories

Given the enormous amount of information that is already available in defect databases, it was a natural step to try to analyze these data to relate the attributes of changed modules to defects reported later during QA activities. Early work in this area, done at IBM by Ostrand led to a predictive defect model and practical defect count predictions. Ostrand studied software systems that were very large, containing 1.2 million and two systems with about 2.5 million executable source lines of code. The source data consisted of (1) defect data collected during system test (for the mainframe system) or during QA (for the PC systems) and (2) change information coming from CM.

7.3. Lessons Learned

A predictive defect analysis was prepared for the Pioneer anomaly journal publication. Many of the results were estimated prior to the communication via four individual C code analyses. The first analysis assumed σv of 0.15 mm/s every five days, and the second assumed an improvement in observational error of ten times, or 0.015 mm/s every five days. These analyses determined the effect on the quadratic acceleration term, σa, and the jerk term, σj. The third and fourth analyses estimated the effects of including one-, two-, and three-directional Doppler data on σa, and the considerations were applied to the residuals obtained. The last analysis also examined the effect of the data span on the estimate of σa.

The final results varied with the assumptions made, but they showed that the modeling of non-gravitational forces induced by heat and attitude control may be amenable to improvement. Should other anomaly-direction hypotheses be put forward, future analyses, either of the same data or of data from another spacecraft, may be able to address them more satisfactorily. This can be demonstrated by examining the estimation of the spacecraft acceleration with four different assumption sets for the residual size, the anomaly direction, and the number of directions into which the data was split. More accurate observational data improves the estimate of the quadratic term by more than an order of magnitude. Also, the amplitude estimate of the constant part of the anomalous acceleration can be improved by a factor of 15.

8. Future Trends in Predictive Defect Analysis

The evolving landscape of predictive defect analysis involves several actors and themes. Commercial software vendors extend software quality tools, employing predictive defect logic to expose hidden risk and searching for better ways to satisfy their customers. Enterprise software developers recognize that business priorities change rapidly and that

every planned release risks ship delays and enhanced costs of nonquality. Metrics and project managers establish budgets and appraisals and look for richer interpretations of historical data. Outsourcing companies explore more effective methods for balancing risks across a project, and subtasking companies seek better ways of relating delivered products to their payments. Maintainers look for heuristics that predict which components will be most expensive to maintain and enhance. Researchers pursue the curves in support of their investigation of influence factors, and professional writers consider the business processes that support better quantitative decision-making.

8.1. Emerging Technologies

Developing predictive defect analysis is an emerging artificial-intelligence-based diagnostic methodology for cryogenic components such as metal-ceramic seals. The goal is to predict helium leaks at room temperature using plasma-spray properties and ceramic microstructural characteristics. Predictive modeling of seal helium leaks is of great interest because it eliminates the need for leak testing at the component level, resulting in reduced manufacturing costs and time. The approach uses a Neural Network (NN) ensemble that combines neural networks in a way that increases the confidence in the results of these state-of-the-art models.

Neural networks are potentially valuable tools for design and manufacturing decision support in situations where the physics of the problem is poorly understood or difficult to model. Plasma-spray process variables, porosity information, and microstructure morphological data provide the inputs to the neural network framework. An ensemble of NNs is constructed using the main prediction as well as the margin of prediction made by these individual NNs. The sensitivity analysis within the NN shows that plasma-spray process variables play a significant role in determining the seal microstructure, which in turn strongly correlates with seal helium leakage. Mathematical frameworks represent the margin of prediction and the confidence in the predicted margin of neural network ensembles.

8.2. Potential Research Directions

Potential research directions in self-organizing hierarchical system monitoring with a focus on automated and adaptive defect analysis are considered. Some promising lines of research are discussed, and the achievable level of adaptability is analyzed.

A stand-alone automated adaptive defect analysis tool is considered; the soundness of an adaptive hierarchical monitoring system equipped with such an analysis engine is discussed. The considerations are equally applicable to various kinds of defects in

hierarchical systems. An area-sensitivity-based visual attention mechanism model forming part of the tool is presented.

9. Conclusion

Predictive Defect Analysis (PDA) is a defect prevention discipline that integrates predictive testing, specification-based testing, exploratory testing, defect classification and analysis, and process change. The goal of PDA is to identify high-risk areas of the application so that appropriate preventive actions can be taken. The capability to predict defect locations and defect counts before the start of formal testing enables the project team to optimize the use of available resources. The process generates defect summaries that, in turn, contribute to ongoing defect prevention projects.

Successful implementation of PDA requires a well-defined process, management commitment throughout the organization, and persistence over several release cycles. Even the simplest implementations add value and establish the conditions necessary to mature toward an optimal implementation. Development and analysis of business process flows and requirements, as well as the corresponding defect data, enhance many aspects of the software development lifecycle. Training resources are widely available, and tools can be acquired or developed internally as needed. In short, Predictive Defect Analysis can realistically become an integral part of the software development mission for many commercial organizations.

References

[1] Runeson P, Andersson C, Thelin T, Andrews A, Berling T. What do we know about defect detection methods?[software testing]. IEEE software. 2006 May 8;23(3):82-90.

[2] Perera A, Aleti A, Turhan B, Böhme M. An experimental assessment of using theoretical defect predictors to guide search-based software testing. IEEE Transactions on Software Engineering. 2022 Jan 31;49(1):131-46.

[3] Kaushik N, Amoui M, Tahvildari L, Liu W, Li S. Defect prioritization in the software industry: challenges and opportunities. In2013 IEEE Sixth International Conference on Software Testing, Verification and Validation 2013 Mar 18 (pp. 70-73). IEEE.

[4] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.

[5] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.

[6] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.

[7] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.
[8] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.
[9] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.
[10] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.
[11] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.
[12] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.
[13] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.
[14] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.
[15] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.
[16] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.
[17] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.
[18] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[19] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.
[20] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[21] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.
[22] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.

[23] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.
[24] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.
[25] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.
[26] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.
[27] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.
[28] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.
[29] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.
[30] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.
[31] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.
[32] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.
[33] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.
[34] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.
[35] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.
[36] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).
[37] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.
[38] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.
[39] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.
[40] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep

cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 5: Intelligent Test Prioritization in Risk-Based Testing Using Artificial Intelligence

Partha Mohapatra

AT&T Corporation

1. Introduction

Tests are important and costly tasks aimed at identifying software defects. Usually, when a new release comes out, fully retesting the system is unfeasible [1,2]. For that test case prioritization techniques propose an order that intends to increase their effectiveness instantly [3-5]. These techniques have been studied over the past few years, and new Artificial Intelligence approaches are utilized through a decision-making algorithm that calculates the relevance of each test into selection to order the tests in an intelligent way. The technique shows its effectiveness on risk-based testing through a case study study in the domain of banking.

The main goal of Software Development is Software Quality. Software Testing is an activity derived for this goal [6,7]. Software Testing is an activity with two main objectives. To invalidate the testing object and to find failures, errors or defects. A failure is when the program runs different than the user expectation; an error is when an action is mistakenly executed or not executed; finally, a defect is the cause of a failure [2,8-10]. Studies have tried to locate the benefits of Software Testing regarding defect detection capability. Software Testing is a timely, costly, and repetitive activity. Avoiding faults while managing the costs are contradictory goals. Test case prioritization aims to order the test execution so that additional testing information is revealed quickly (for example, faults). The prioritization process must select and execute tests with the maximal value. In addition, the output is an ordering of new and old tests so that faults can be successfully identified [1,11-12]. An intelligent test case prioritization approach combines fuzzy risk-based testing and test case prioritization to re-execute test cases that are loaded with the highest risk during each new release. The technique suggests an order

of execution for test cases such that the additional information obtained as early as possible during regression testing fulfills the requirement of risk-based testing.

2. Understanding Intelligent Test Prioritization

Testing is a crucial phase during the software development life-cycle and substantially contributes to quality assurance. Risk-Based Testing (RBT) prioritizes testing of requirements based on their associated risk. It is essential to consider two risk factors for an RBT model: the probability of failure and the business impact of failure. In determining the test execution schedule, utilizing the AI portion of the risk can be regarded as an intelligent test prioritization in RBT.

Risk-based testing (RBT) estimates the quality loss of an application and assigns test priorities based on potential risk. However, current approaches have certain limitations. For instance, they consider only specific risk factors and overlook others; for example, many models focus on the business impact of failure and ignore detection or probability of failure. Intelligent test prioritization (ITP) aims to predict risk associated with each test case and then schedule the tests according to their risk values. Together, RBT and ITP provide prioritized test schedules that accommodate the software's risk profile. A variety of approaches—AI-based, model-based, and tabu search-based—have been used for test prioritization. The efficiency of these scheduling approaches depends on the nature of the dataset and the associated risk factors. Although numerous risk-based test prioritization (RBTP) models have been developed, none of them effectively addresses both substantial and moderate risk. Consequently, the present approach involves designing an RBTP model that applies AI at the probability-of-failure level. This model prioritizes tests such that the cumulative risk of high-risk functions increases rapidly at the outset, followed by moderate-risk functions.

3. Risk-Based Testing Overview

Risk-based testing (RBT) prioritizes testing activities based on the residual risks associated with software components [13-15]. The residual risk of a component is the risk that remains after it passes testing and is determined jointly by the component’s exposure and the customer’s aversion to it. In many systems, after risk assessment maximizes the coverage of the produced exploit kits, it prioritizes the most exposed components using an automated algorithm for exploitation scoring. Neural networks take the input of risk factors and output the precise classification of the residual risk.

Current RBT systems use risk matrix grids to assign the priority of testing activities. Risks are assessed using the current security state based on three risk factors: impact,

probability, and detectability. Both impact and probability of security risks can be broadly determined using two considerations. First, the execution environment of the component, which includes its deployment infrastructure (the affected channel) and the business of the hosting organization (the affected business), is identified to be either of high, medium, or low level [16,17]. This assessment is mainly based on the skill and resources of the adversary who can craft the attack, the privacy level of the data affecting customers, the public safety issues, and the system function at a given channel state. Next, the component is located in the organization's network and its required connectivity to the other components within the system or to the outside world is identified. The component cannot survive without the needed connectivity. These three considerations can guide testers on estimating impact and probability, although it is still challenging to assess the exact level of impact or probability.

3.1. Definition and Importance

Risk-Based Testing is an approach that uses system or operational risks, or an early risk analysis, to guide the testing process in order to perform risk mitigation [12,18-20]. It allows prioritizing testing, and focuses the testing on product components that have a higher impact on the product quality. Risk-Based Test prioritization consists of using the early test risk assessment to schedule test cases execution in a way that reduces hazard impact or fault probability as soon as possible. Hence, the availability of a reliable early test risk assessment is crucial for defensible decision-making in testing. New Risk-Based Test Prioritization approaches based on Artificial Intelligence are able to {1} continuously and automatically evaluate the testing coverage associated with a risk of the hazard, and {2} use this risk coverage level to schedule test cases execution in a way that reduces hazard impact or fault probability as soon as possible.

Testing is expensive and hardly ever guarantees the absence of faults in the final product. Risk-based testing, usually adopted for mission critical systems, uses information about faults' criticality and likelihood of occurrence to focus the efforts on the most critical warnings. Artificial Intelligence models can learn from previous faults and use this information to predict the criticality of the artifacts being faulted, as well as the likelihood of their occurrence and detection. By using these values, it is possible to identify the faults with higher risk levels and include them in the test priority list. The risk formula adopted is: Risk = (Hazardousness × Fault Probability) / (Detection Probability × Test Coverage).

3.2. Key Principles of Risk-Based Testing

Many test automation concepts and frameworks enable the use of risk-based testing. The most essential principles of risk-based testing, which are decisive for future research and development in the field of test prioritization in risk-based testing, are listed here [21-23]. Risk-based testing is based on the assumption that the testing-process resource constraints and the residual risk resulting from remaining-untested aspects of the product are balanced. Risk-based testing requires hierarchical decomposition of the analysis models and estimates of risk at appropriate levels. Risk-based testing supports the continuous refinement of risk estimates. It allows and supports the incorporation of virtually any desired type of risk estimation method. Finally, risk-based testing provides explicit, verifiable risk-mitigation decisions. Risk estimation is inherently imperfect. Risk-based testing supports decision-making despite risk assessor imperfection.

More is not always better. Executing more tests does not necessarily translate into a more risk-mitigated system [24,25]. It matters what tests get executed. The fundamental principle of risk-based testing is the balance between two aspects of testing: “test resource expenditure” and “risk mitigation.” The risk associated with the Un-tested functionality is represented by the residual 30 risk model. The risk associated with the Testing effort is represented by the test-resource-cost model. The relationship between the two is represented by the risk-mitigation model. As in any field, it is critical to test the type and prior-ity of tests required to support the mission. Both the mission and the environment are continuously evolving, and test resources traditionally have been bro-ken into buckets to protect groups of individuals: there are developers tests, typically unit tests that test each component in isolation and that provide critical feed-back for the development and maintenance of the component; there are system tests, which test the system as it is integrated, looking for enforced requirements, and checking the overall safety of the system; and finally acceptance tests, which include all the money or capital tests and are testing the system that is ready to be deployed, that is going to be put in the operation of the company. That testing that really tests that the requirements are fulfilled, the business requirements of the company, and the system do what it is designed for and trustful.

4. Role of Artificial Intelligence in Testing

Artificial Intelligence helps testers by analyzing available information and making predictions [26-28]. AI enables test prioritization and selection based on predicted risks. When old test results on new program versions are preserved, they form predictive models capable of indicating which tests should be executed first or not at all. Risk-based testing methods anticipate risks and define test cases accordingly, directing testing efforts toward the most critical aspects of the software product. Risk-based testing relies

primarily on heuristics to estimate risk levels; however, Artificial Intelligence can take risk assessments a step further by training a Risk Prediction Model on previous risk analysis data. Before executing the entire test suite, the Risk Prediction Model can identify tests related to classes with the highest predicted risk levels, enabling testers to address potential problems more promptly.

Selection of test cases for each build is susceptible to missing faults due to software modifications, as several paths and features remain untested [29-31]. These risks can have a greater impact on the final product. To address this risk, test case selection for each build must be risk-based and utilize the testing efforts more effectively. Testing can be made more efficient by ordering the set of test cases according to the likelihood of fault detection. Politics of toptest selection can not address both selection and ordering of test cases but few papers address priority of test cases to reclaim of the finished projects. Implementing effective test case prioritization methods can make the software testing phase more efficient and influences the defect finding capability of a test case set.

4.1. AI Techniques in Software Testing

Many papers in software testing confirm the usefulness of AI techniques—especially Machine Learning (ML)—for various testing activities and the overall Software Testing Life Cycle (STLC). AI techniques proved helpful in numerous testing activities, for example, in filtering requirements, in Object-Oriented testing, in risk-based testing and in test automation (Section 4.4). Similarly, ML has been used to predict the testing effort, for example, when having given specifications or during the development of test cases. Additional uses include bug prediction, estimating the testing efforts for test case prioritization, risk assessment in risk-based testing, and generating test inputs to support automated testing. Hence, it is evident that AI techniques such as ML have already been employed in different testing activities in the STLC. Artificial Intelligent Test Case Prioritization Currently, there is little research employing AI techniques for test case prioritization. Some papers use Artificial Neural Networks (ANNs), Logistic Regression, and Decision Tree models to establish suitable testing orderings for Agile Test-Driven Development. Other authors propose the combination of genetic programming and fuzzy systems to prioritize tests in risk-based testing. A different approach applies an ML technique to determine the most suspicious assertions within test cases suitable for regression testing.

4.2. Benefits of AI in Test Prioritization

Risk-based testing is a method that uses risk analysis to prioritize testing, reducing the level of exposure to a given risk and speeding up risk detection [3,32,33]. Test cases executed earlier produce results that are suitable for fixing the most important faults, which leads to general productivity improvements. Test prioritization is one such strategy that aims to fulfill a testing goal, whether it is to detect faults earlier, enhance coverage, or reduce the total waiting time. It reorders an existing suite of test cases to meet that goal. Test prioritization techniques can use no information at all, only historical execution data, or even product information. Test selection techniques, on the other hand, choose a subset of test cases to fulfill a goal. AI algorithms can be applied in any of these strategies.

5. Impact Analysis through Machine Learning

While the prioritization of test cases is a well-studied area in risk-based testing, test selection and related impact analysis based on artificial intelligence and machine learning is less addressed [4,34-36]. Previous research in risk-based testing applied fuzzy inference, initially presented for risk assessment, to the topic of test selection. More recently, a dynamic classifier was used to identify the risk category of a bug report, thereby supporting impact analysis and enhancing efficiency.

Within the framework of risk-based testing, input from diverse sources is consolidated in the risk assessment phase. One focal point is the estimation of risk exposure. To quantify the risk exposure for a requirement, a machine-learning classifier was trained with requirement data from past projects and failure classification data from the final project phase. Following the categorization of all requirements according to risk classification, the requirements are ranked in descending order. This ranking identifies the requirements associated with the highest risk exposure situation and facilitates the prioritization of corresponding test sets in the test planning phase.

5.1. Machine Learning Fundamentals

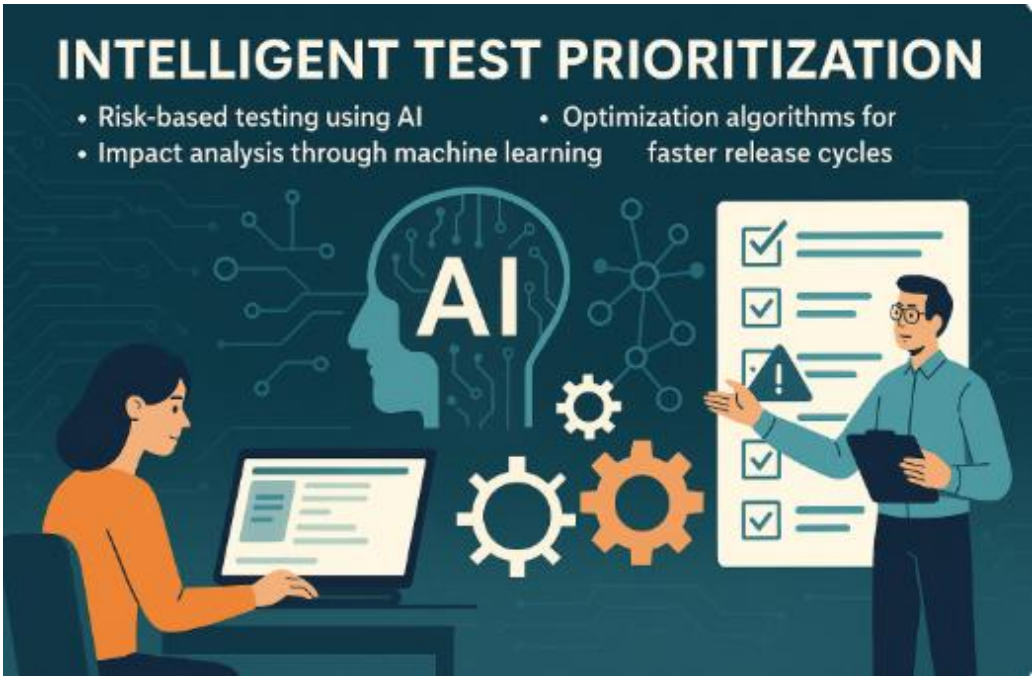
The area of Artificial Intelligence, especially Machine Learning, is becoming more popular in the testing field. Today, Machine Learning itself is a very abstract and umbrella term. Broadly speaking, Machine Learning is a given subfield of AI and describes the training of a specific system with given data so that it can make certain automated decisions when requested based on this data without a direct human impact. The decision depends on the specific subfield described in the section below.This work mainly focuses on Supervised Learning.

Supervised Learning describes a subfield of Machine Learning where a model tries to infer a mapping function f from given input–output pairs (Xi, Yi) such that desired output values of new input values can be predicted. The model learns from known images of for example cats and dogs, i.e., X = {images} and Y = {labels} to later categorize given new images as either cat or dog.

A typical process to create this model involves several steps. First, the given input data (e.g., images) need to be represented in a structured table format with a fixed number of columns. Each column contains a certain interpretable "property" of the training data that can be converted to a vector representation. Next, a Machine Learning algorithm is selected, such as Logistic Regression. Then, known inputs Xtrain and their corresponding labels Ytrain are given as the training data. Based on this, the model internal parameters are optimized, and afterwards Ytest labels for previously unknown inputs Xtest are predicted. These steps are summarized in Figure 5.1.

5.2. Using Machine Learning for Impact Analysis

Other interesting work for risk-based test prioritization using impact analysis can be found in Anderson and Hayes's ''Machine Learning Driven Impact Analysis for Risk-Based Testing,'' which investigates the feasibility of using machine learning in risk-based testing, focusing primarily on impact analysis. The key idea in their work is that a more intelligent risk-based testing process can be developed by leveraging the information contained in previous testing efforts and using machine learning classifiers to perform impact prediction. In this way, impact analysis can be automated, enabling more intelligent and realistic risk-based test prioritization [37-40]. To analyze the support that impact analysis for risk-based test prioritization can expect from machine-trained classifiers, the authors adapted the ''Test Case Prioritization Using Risk Metrics'' algorithm, using the output of the classifiers for the change impact prediction. Test cases that cover methods classified as impacted have a higher priority. Thus, a better classification leads to better test prioritization. The outcome of the analysis indicated that machine learning classifiers can be used for impact analysis. The approach replicates human judgment in impact analysis by learning from decisions made for previous changes and utilizes these lessons for impact prediction of future changes.



5.3. Case Studies on Machine Learning Applications

At the Institute for Software Engineering and Testing of the University of Stuttgart, researchers have developed techniques for risk-based testing in which a test manager estimates the risk that a test case fails, and then uses this estimation to prioritize the tests for a test run. To reduce the manual effort involved in estimating the risk levels, the researchers used machine learning to learn from previous estimations and automatically classify new test cases. To evaluate the performance of several classifiers, the researchers carried out laboratory studies with students, as well as case studies in which three industry practitioners performed the risk-level estimation and the learning was based on their inputs. The results indicated that the classifiers can reduce the estimation effort by an average of 50% while maintaining a satisfactory performance.

The German IT service provider GFT carried out several projects to evaluate the applicability of AI technologies in software engineering. One project investigated the use of machine learning to infer risk levels for the test prioritization of a banking application [6,7]. The learning was performed by training an artificial neural network with data from previous test runs, and two case studies were conducted that used GFT employees as risk assessors. The results also confirmed that the company could save approximately 50% of the estimation effort without severely compromising performance. Both studies are detailed in the cited literature.

6. Optimization Algorithms for Faster Release Cycles

An overview of optimization algorithms explains their purpose, selection criteria, and implementation process, focusing on their capacity to accelerate testing workflows. The discussion considers earlier concepts of artificial intelligence and machine learning to maintain a cohesive narrative. Intelligent test prioritization in risk-based testing speeds up software release cycles. Impact analysis of changed requirements—using software risk information—in conjunction with optimization algorithms significantly shortens testing time compared to conventional risk-based-testing methods. Experimental results demonstrate improved fault-detection rates and, consequently, reduced testing durations achieved through the impact analysis of requirements coupled with suitable optimization algorithms.

Complex software systems undergo constant change—through new adaptations, additions, and enhancements of existing requirements—to keep pace with competitors. During the testing phase, time constraints often lead testers to overlook some test cases, resulting in customer dissatisfaction after deployment. AI techniques are used to address the test-case-prioritization problem. Incorporating an impact-analysis step in risk-based testing further accelerates the testing cycle by enabling the prioritization of only those risk-prioritized test cases that are affected by changed requirements.

6.1. Overview of Optimization Algorithms

Optimization algorithms offer a flexible approach to address combinatorial optimization problems in software engineering. They automate the search for optimal or near-optimal solutions, accelerating risk-based testing and enabling quicker public releases. Consequently, selecting an appropriate optimization algorithm for test prioritization is crucial. Standard algorithms include greedy algorithms, genetic algorithms, and particle swarm optimization algorithms. Greedy algorithms select the best available solution at each step, though the choice of objective function can be challenging, and they risk converging to local optima.

Genetic algorithms are based on Darwinian evolution principles and natural selection. They typically commence with a randomly generated population, with the functions for population generation, stagnation, mutation, and crossover remaining constant throughout the process. The best fitness function value up to any generation becomes the global optimal solution. Particle swarm optimization algorithms draw inspiration from natural examples like birds foraging.

6.2. Algorithm Selection Criteria

In the candidate-list generation step, a ranking criterion is needed to select the best candidate list. It requires an appropriate algorithm-selection decision model determining the test case sequence with shortest total test execution time. Selecting an appropriate method is of special importance due to the varying performance of the algorithms on different instances of the problem at hand. Moreover, the time limits of the testing phase often prevent the usage of methods with very high runtimes, such as metaheuristics.

For the shorter requirements sequence, it is assumed that a test-case sequence covering the requirement sequence can be generated using a straightforward heuristic. As soon as all required actions of the first m requirements have been executed, the test ends. Under certain conditions, the Shortest Processing Time (SPT) dispatching heuristic relative to the requirement costs of the pending requirements generates the optimal shorter test case — namely, when the execution times of the test cases are the same. However, these assumptions do not allow modeling shortcomings of requirements or test cases. A more versatile model is provided by the Makespan (MS) criterion, which minimizes the duration of the testing phase. There are two promising metaheuristics for minimizing the makespan in a permutation flow shop — tabu search and genetic algorithms. Tabu search is an iterative metaheuristic that uses a heuristic to explore the neighborhood of the current solution by selecting the best possible solution even if it is not better than the current solution. Genetic algorithms are population-based metaheuristics that use mechanisms inspired by biological evolution to iteratively improve a population of candidate solutions.

6.3. Implementing Optimization in Testing

Optimization refers to the process of making something as functional or effective as possible, often with respect to speed or cost. In the context of testing during the software development lifecycle, it involves applying certain strategies to achieve the fastest possible testing execution. As a formal technique, it is the process of finding the best decision, considering available options or alternatives.

Applying optimization to testing during a project targets risk-based and intelligent approaches. It aims to find the fastest test suite implementation while maintaining test case quality and coverage. User control allows for establishing constraints or limitations on selection criteria variables, such as the maximum number of test cases or the maximum acceptable execution time. Such constraints impart realism to the results, making the fastest suite development possible under the defined conditions. Optimization may consider numerous variables; the primary focus here, aligned with risk-based testing, is on prioritizing testing based on the impact of requirements changes.

This aspect demonstrates how knowledge of impact-related information can make testing execution more efficient.

7. Integrating AI and Machine Learning in Testing Frameworks

AI and Machine Learning techniques have largely been ignored in the context of Risk-Based Testing, such as Intelligent Test Prioritization. In two recent papers, a general framework for Risk-Based Testing has been introduced and general aspects of AI in testing have been reviewed. The following discussion uses parts of this framework together with a review of AI in software testing to describe how Intelligence risks in the context of Business Intelligence and Machine Learning can be handled by means of Intelligent Test Prioritization. The goal is to propose a unifying framework that aids the integration of AI in Risk-Based Testing and Test Prioritization, enabling the intelligent prioritization of tests for Intelligence Risks.

Operational Risk Management is an often-neglected area of Enterprise Risk Management and focuses on the identification and evaluation of risks arising during the daily operations of a company. Operational compliance risk deals specifically with the operational risks that arise concerning the regulations a company must comply with. Business Intelligence is responsible for providing information that helps companies reduce these risks. However, this information is no silver bullet, as it can itself be a significant source of bias, skewing the perception of risk and leading to incorrect assessments. This aspect is especially important in the operations of risk-oriented regulatory authorities, which are responsible for monitoring the financial institutions and insurance undertakings that have to perform operational risk management. Misconceptions regarding the nature of risk thus hold the potential to have an impact on society at large. Machine Learning techniques, often used in the analysis of Operational Risks, are particularly vulnerable to subtle forms of bias.

7.1. Framework Design Considerations

Lossless prioritization of test cases in risk-based testing aims to find an optimal ordering with respect to the risk metric. Finding such an optimal solution is a computationally hard problem and an order width of only 3 with pruned test suites makes the problem still NP-hard. Several prioritization techniques applied to risk-based testing exist, out of which incremental prioritization is reviewed in more detail. In incremental prioritization, the first test case selected is the one with the highest risk. At each step, the test case with the highest risk of the remaining test cases is chosen and added to the prioritized list. For risk category coverage techniques, the same mechanism is applied but in this case, the category with the highest risk is selected first.

Artificial intelligence can be used to automate the test case prioritization and to suggest tests according to different selection goals. A tool implementing this technique supports incremental prioritization and considers several prioritization goals, such as the maximization of risk or mutation score or the maximization of risk or mutation category coverage. It uses genetic algorithms for the prioritization of a test suite with more than two hundred tests, pruned according to multiple risk criteria. Using the risk for detection of faults as prioritization criterion allows Klee to perform a lossless prioritization of the pruned test suite.

7.2. Tools and Technologies

Due to its AI past, the tool was implemented in Python, which hosts a massive ecosystem of AI and DL-specific libraries and tools. It is portable, lightweight, and renowned for being one of the most readable programming languages.

Detectron2 grants the project access to a variety of state-of-the-art, pre-trained models that are the basis of the object detection and classification model. It is an object detection and segmentation framework implemented in PyTorch and was developed by the Facebook Artificial Intelligence Research (FAIR) team. It can be used for a wide range of computer vision applications. PyTorch was selected over any other machine learning framework because of its dynamic computational graph, high-end capabilities, simplicity, and Pythonic nature.

8. Challenges in Intelligent Test Prioritization

With increasing software complexity and tight releases with limited time, testing of large software projects cannot be done exhaustively, within the limited time available. Therefore, in production, testing is often done in a prioritized way. In a risk-based test prioritization system, important or more risky areas of the software are tested first and less risky areas are tested towards the end. Priority for an area of the product is assigned based on the possible effect or impact of a failure in that area of the software product.

Risk mitigation is, therefore, a basic tenet of software testing. Non-risk-based techniques do not consider the severity of any area of the software for selecting tests but use other information to select tests, such as dependencies, access to source code or knowledge of the source code. These techniques are, therefore, less effective in addressing risk associated with failures due to increased time and incomplete coverage.

8.1. Data Quality and Availability

The quality of the risk model strongly depends on the quality of the risk factors. Business experts should estimate the individual risky features by setting a maximum and minimum value as well as an expected average for each risk factor. Subsequently, test designers rate the test items regarding the risk factors by assessing the position of each test item regarding the corresponding risk factor. Depending on the project progress, the most actual category can be selected.

Depending on the criticality and riskiness of the SUT and the test strategy, the quantity and quality of the available testing information that can be used are different. The number of available test cases (items) is limited in the early and medium test phases because only the atomic or integration tests are implemented so far. In the last phase, the number of test cases considerably increases for system-level testing. There is usually a shortage of information during the early phases of a release, whereas in the final phase, a variety of information can be utilised both for new test cases and existing historical test cases. Therefore, it is advisable to apply different test prioritisation approaches depending on the progress of a test phase or product.

8.2. Scalability Issues

From a theoretical perspective, evolutionary algorithms have very weak scalability compared to human testers. Evolutionary algorithms are exact-search algorithms. In each iteration, they perform searches in the way of local neighborhood in state space. If the local optimum of the search becomes too deep, the evolutionary algorithm easily falls into a local optimum. The use of a local neighborhood also makes the evolutionary algorithm extremely OpenAI ChatGPT Zoom poor at parallel operations. It can only perform one local search at a time. Thus, it is inevitable that the evolutionary algorithm will spend excessive time searching in the same area of the test cases by repeating and swiping.

On the contrary, human testers can perform a global search on the test case space by combining knowledge. For example, a tester can first locate the suspicious test cases in the test case space and then locate similar test cases by using the test case properties. In this way, the tester can converge to the high-risk test cases. What's more, because of the knowledge-based global search, human testers are very easy to perform multi-threading global searches and compare the local optimum of multiple threads to jump out of the local optimum.

From an empirical perspective, it can be found that in general, the performance evolvability of AI-based test case prioritization is lower than that of human designers. In the case of prioritizing the same number of test cases, it is very difficult for the EA to produce test cases with a higher degree of risk than the human-designed test cases.

One possible solution is to enhance human intelligence with AI. For example, the AI algorithm can provide guidance for human testers to efficiently select test cases for prioritization.

8.3. Resistance to Change in Testing Practices

Resistance to change is one of the reasons why organizations have difficulties to adopt risk-based test management in practice. Nonetheless, the use of risk-based test management enables a much more precise measurement of the residual risk of a software system. Moreover, in combination with test prioritization, the highest risk is mitigated first. This twofold use of risk-based test management in testing calls for an approachable mitigation of the negative memories of the past.

In complex IT environments, enterprises usually have a large variety of applications for different purposes. Accordingly, for system and acceptance tests, there are test cases without clear requirements, test cases that have lost their connection to actual IT faults, test cases that have been duplicated multiple times because connections have changed over decades. Whenever similar test cases exist, these numerous conditions are typically all checked, even if some of them are less decisive for the applications—whether because of their size, whether that is a large number of microservices, or because of their economic value, the test cases are created.

9. Future Trends in Test Prioritization

Current test prioritization approaches are slowly on the way of being supported and at some degree even replaced by the application of AI. The application of AI in supporting testing activities currently present several research directions. Testing activities do not typically require a big data approach. However, to provide AI applications support, the achievement of one of the characteristic of big data, which is the availability of a huge volume of data, is crucial. This is because the accuracy of the AI-supported testing activities depends on the input data through the learning, or training, phase, which directly affects the quality and the source of the data.

The implementation of AI in testing activities, in the course of the provided service, employs several AI techniques, such as artificial neural networks, naive Bayes classifiers, fuzzy logic, genetic algorithms, support vector machines, and decision trees. Common testing activities considered, according to the categories presented in the taxonomy, include prioritization, selection and minimization, test case generation, test automation, performance testing, test evaluation, and attempt to describe the AI-support available for these activities address the risk-based testing process framework. In the

context of risk-based testing, within the test planning activity area, AI support is available for risk driver determination, risk item identification, risk item estimation, and test strategy determination. Within the Test Design & Implementation activity area, AI support is available for test evaluation and attempt to describe the AI support available for these activities.

9.1. Emerging Technologies

AI-based techniques have demonstrated remarkable performance in many domains, e.g., the arts, natural language processing, voice recognition, and game playing. These recent advances generated considerable interest in AI also in software testing, where AI-based testing is envisioned to help automate testing activities and produce better test assets. However, although AI-based testing has attracted increasing interest, it is still an emerging field. Current research focuses on the use of AI to generate test cases (including model-based testing), the classification of test items to support testing activities (including risk-based testing), and the selection and prioritization of test assets (again, including risk-based testing). In general, AI-based testing has demonstrated effective and efficient risk classification in risk-based testing. Among all risk-based testing risk analysis methods, fuzzy expert systems are an emerging topic, especially in test case prioritization.

A novel fuzzy expert system application is presented for test case prioritization in risk-based testing (REPT). Risks associated with each test case are assessed through a linguistic scale provided by a set of risk factors pre-classified by experienced testers. All risk factors are involved in both test case and risk judgment lying upon a fuzzy inference engine containing fuzzy if-then rules with membership values of Low, Medium, and High. The determined risk level is then used to prioritize test cases before executing those with the highest risk level. The system is evaluated in comparison with an existing risk-based test case prioritization (RPTCP) method, combining Pathfinder. Experimental results manifest the effectiveness of REPT; test cases are allocated into distinct level sets in which the execution of high-level risk test cases precedes low-level risk test cases.

9.2. Predictions for AI in Testing

How intelligent test prioritization can be achieved by using artificial intelligence to support risk-based testing is explained in detail. Accounting for the two risk factors, likelihood and impact, for which test cases should be prioritized, it turned out that the best way to estimate the likelihood of defects in software updates is an approach sentence similarity. The training results also showed that both the textual information of the RTS and the RTS change descriptions should be used. Concerning the impact risk factor, a

decision tree-based impact-level prediction model, in which model variables from the RTS, the underlying Issue Tracking System (ITS) and software repositories are combined, achieved the best results.

An approach that combines these results to achieve comprehensive intelligent test prioritization as well as the most important research directions in this field are the subject of further research. The best performing models for both risk factors should be combined to create a comprehensive likelihood-impact model. In addition, further risk factors should be identified and integrated into the model. To take full advantage of the benefits that machine learning can offer in test management, the development of an integrated and continuously updated risk estimation model for intelligent test planning remains a challenging research topic.

10. Conclusion

Test prioritization constitutes a vital technique to elevate the efficacy and reliability of risk-based testing. The immediate objectives of test prioritization encompass not only the scheduling of risk-related tests that have yet to be executed but also the recommencement of testing as swiftly as possible following any interruption, so as to minimize risks. The diverse factors examined encourage the use of human knowledge and guide the selection of classification algorithms; the risks revealed influence the choice of data mining algorithms.

Low-risk tests are generally evaluated with more leniency in terms of execution time compared to high-risk tests. Intelligent test prioritization advocates for the incorporation of artificial intelligence to machines, enabling them to learn from the repository of testing knowledge and thereby execute testing tasks with enhanced precision and efficiency. While traditional risk-based testing predominantly emphasizes risk analysis, transitions, and mitigation, intelligent test scheduling also encompasses the prediction of average execution time, average cost, and the probability of scheduling interruption. These additional factors offer a more expansive perspective that aids organizations in the effective scheduling of risk-based testing.

References

[1] Baqar M, Khanda R. The Future of Software Testing: AI–Powered Test Case Generation and Validation. InIntelligent Computing-Proceedings of the Computing Conference 2025 Jun 19 (pp. 276-300). Cham: Springer Nature Switzerland.

[2] Li Y, Liu P, Wang H, Chu J, Wong WE. Evaluating large language models for software testing. Computer Standards & Interfaces. 2025 Apr 1;93:103942.

[3] Lönnfält A, Tu V, Gay G, Singh A, Tahvili S. An intelligent test management system for optimizing decision making during software testing. Journal of Systems and Software. 2025 Jan 1;219:112202.
[4] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.
[5] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.
[6] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.
[7] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.
[8] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).
[9] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.
[10] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.
[11] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.
[12] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.
[13] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.
[14] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.
[15] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.
[16] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.
[17] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.
[18] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.
[19] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.

[20] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.
[21] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.
[22] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.
[23] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.
[24] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.
[25] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.
[26] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.
[27] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.
[28] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.
[29] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.
[30] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.
[31] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.
[32] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.
[33] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.
[34] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.
[35] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.

[36] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.
[37] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[38] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.
[39] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.
[40] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 6: Artificial Intelligence in Test Automation and Self-Healing

Partha Mohapatra

AT&T Corporation

1. Introduction to AI in Test Automation

AI in test automation is the application of Artificial Intelligence to automate the testing and verification of software products [1,2]. This includes the creation and maintenance of test scripts, execution of test cases, and diagnosis of any anomalies within the testing process or the application under test (AUT). The struggle with software testing is that it is a laborious, time-consuming, and sometimes "boring" process. By introducing AI and subsequently implementing AI test automation, many of these challenges can be eliminated [3-5].

AI can simultaneously improve both the process and coverage of testing. It can design test inputs cleverly and mine test logs to discover failures and their root causes—tasks that are often prohibitive for traditional, manual, or exhaustive techniques. Furthermore, AI has the ability to "learn" from production data and automatically generate new test cases based on real business use. AI can also create or recommend test data by translating test requirements to natural language, selecting production data using AI strategy, and modifying test data to meet >12 data requirements. It can be used to monitor the testing process, identifying which parts can be automated, which parts have a higher possibility of having bugs, when risk mitigation testing needs to be completed, and when all risk management is successfully completed.

Self-healing within the test automation domain leverages machine learning and/or natural language processing to reduce the time required to maintain test scripts [2,6]. Test automation monitors the health and metrics of automated software tests and provides insights into the quality of the software product and the efficiency of test automation. According to the "Test Automation Maturity Model" provided by the

organization Test Automation University, maintaining test automation is the most costly and resource-intensive part of test automation.

2. The Concept of Self-Healing Test Scripts

The association of terms like “AI” or “Artificial Intelligence” with test automation evokes a multitude of connotations among professionals [7-9]. While some anticipate the development of shapely, humanlike robots capable of autonomously executing all testing tasks, others foresee the emergence of self-healing test scripts driven by AI.

Before delving into the discussion on Artificial Intelligence potentially addressing the maintenance and enhancement of test scripts, it is imperative to comprehend the process itself. Self-healing Test Scripts pertain to the automated identification of invalid objects within a test script, followed by their rectification through AI techniques and algorithmic methodologies, all occurring without any involvement from the tester of the Test Automation Framework, thus constituting a fully automated task.

2.1. Definition and Importance

Test automation (TA) is concerned with using software bots, also known as test bots or test scripts, to automatically execute testing processes. It is a widely used activity in

software development that offers substantial advantages to developers. For example, automation reduces time, effort, and costs associated with manual testing processes. In addition, the software testing process benefits because it is free from errors caused by human mistakes.

However, because of changes to Application Under Test (AUT), broken nature of test scripts may impact the continuity of the testing process and reduce the benefits of automation. A typical automation testing process includes tests written in a preferred programming language, followed by execution on the application under test. As it is a non-trivial task to maintain and update test cases to keep them in sync with the development of AUTs, current approaches address this challenge by incorporating self-healing mechanisms in test scripts using artificial intelligence. Self-healing AI Auto Click Testing Tools make test scripts self-healing by incorporating AI. The main goal is to generate or update test automation scripts automatically whenever a change has occurred in the layout of an application.

2.2. How Self-Healing Works

The self-healing function is a form of artificial intelligence that ML and NLP enable. TestAutomation.ai created the first self-healing tool in 2017, and the functionality is now part of many testing tools [10,11]. Commercial tools offer it for Web, API services, mobile, SAP, and Oracle applications.

When an element is modified or removed from an application, self-healing first checks whether the test object has been manually updated recently [12-14]. This practise reduces the risk of the system overwriting a recent manual adjustment with an automatic update. When the element has not recently been modified, test remediation determines what actions to take to execute the test case. Self-healing can access the application most recently tested using the service API or can create a sandbox or development environment and deploy the most recent build. This capability is important because it enables the use of dynamic data, such as a booking ID or order ID, from the current test and helps to correlate the test.

2.3. Benefits of Self-Healing Test Scripts

Using AI in automation testing provides many benefits, including faster creation of test scripts, improved maintenance of test scripts and overall reliability of software. Test scripts break often, mostly due to constantly changing user interfaces of the software being tested and changes in the test data and/or connectors to third party software.

Test failures may be a result of bug in the software being tested, or due to bug in the test automation scripts (not designed correctly or not maintained). It becomes difficult to determine in which area the failure had occurred and hence takes more time to evaluate and provide feedback. The time taken to evaluate causes of failure cannot be used to create new test scripts or execute additional testing to further improve the quality of software. This leads to delayed feedback from the testing teams and delayed fixes from the development team, impacting the overall release of the software.

Self-healing gets triggered when the test scripts are broken either due to changes in the objects in the UI or test data or third party services, connectors points to open source and proprietary convergent systems. Self-healing tests are typically triggered when the standard execution engine finds an exception during execution. Self-healing logic acts as an overlay on the standard execution flow and can be invoked either as a separate thread when an exception occurs or as a separate execution after the failure of the execution passes.

3. Adaptive Locators with AI

Adaptive Locators with AI Explaining the term Adaptive Locators with AI employed in test automation techniques [3,15-17]. Who are interested in AI testing and self-healing test scripts can use the given information. Self-healing test scripts refers to the concept, if the test script breaks due to any change in application or due to failure, there is an automated script correction it can resolve the failure of test. How it can be done? – By using intelligently Adaptive locators with AI. When application is a static application it is easy to be done by the normal synonyms used in test automation. Any changes are done then change it in the application, the same way scripts need to change. When the application is dynamic locators are also dynamic then it is very difficult to maintain those test cases. Applications look and feel may be the same but the properties will change on the every releases. How do we automate the locators where application properties are dynamic in nature? How do we solve the self-healing mechanism? How do we automate this locator change in script?

3.1. Overview of Adaptive Locators

Loss of locator is one of the common reasons behind failure in the automation testing script. Maintaining these causes a lot of effort. A frequent change in application underlying code creates a lot of work for automation testers or sometimes automation architects. The concept of Adaptive Locators is used to heal locators whenever the code is changed automatically and makes the test automation script executable without any failure or issue when executed. Machine learning technique can be used to make the

locator adaptive [18-20]. Various elements of the application are identified properly by the AI model and all attributes of the element are considered as features and trained by the model. When the change is made in the underlying code and when the testing automation script fails because of the locator not found, the AI model predicts the next best possible locator as per the training it already has.

Regeneration of test-case script is costly and error prone [21-23]. It is quite important to update the existing test case scripts. Self-healing test scripts automated the update of the test case scripts by themselves whenever they fail. WysiWyg AITM automation features from Accelrysai responded to the user preference of self-healing behavior in testing. Using the self-healing capabilities offered by AI, these bots eliminate the repetitive steps followed in testing. Testing involves executing the test script repetitively and the automation script frequently faces the failure due to change in the code. These failures trigger the activity of recording a new test-case script which is time-intensive. Regulators also expect automation testing to be performed for each release of the application which demands frequent execution of the stale test-case script and increase the cost of maintenance. Self-healing bots enable automation testing to be performed in a better way for the release of an application.

3.2. AI Techniques for Locator Adaptation

Element locators are crucial in determining the status of web elements during automated test execution. In any of the supported browsers, the current status of an element can be obtained easily using the current locator [9,24,25]. This information, however, cannot be used directly to compare with the test case in order to find the pass or fail state of the test. Due to changes in applications between releases, the actual element status of the web element will differ from the testcase status. Test failures due to such changes are misleading to the developers under test. Adaptive locators handle this problem by automatically matching the element status derived from the actual application to the expected element status mentioned in the test case, thereby making the test case "self-healing." During matching, if any element is found to be resembling the "By" locators more closely (for example, in case of projects where enterable magnitude is dynamically changing), the "By" locators are rebuilt and this changed "By" locator is used for execution of further steps of the same test case.

All possible attributes for the actual status of the element will be retrieved from the execution screen and a matching score will be generated for the element by applying Artificial Intelligence techniques such as self-attention and natural language query-based machine learning models, depending on the user preferences. These algorithms will identify the element that most closely resembles the "By" locator, and if the score exceeds the twenty-five score threshold, it will be considered a match. This score

threshold is derived from the extensive empirical results of the team of authors and experience of building, maintaining, and enhancing adaptive locator-based tools. The "By" locator is then rebuilt and recorded for next time use.

3.3. Case Studies on Adaptive Locators

Adaptive locators have proven their capabilities in many case studies conducted for AI in test automation [26-28]. In one study, Joseph found that adaptive locators reduced the requirement for locator maintenance—an Encore topic. The approach, employing a Random Forest classifier, was built on features extracted from the DOM structure such as 5-gram and string-based features. An internally developed framework utilized the Document Object Model's accessibility tree and HTML tag attributes, applying resurrection methods like static, dynamic, and AI-based locating techniques. Notably, AI locator techniques offered the highest repair success rates and lowered flakiness in Selenium test scripts.

In the self-healing domain, Thakur showcased that self-healing scripts could fulfill user expectations. In another report, it was demonstrated that self-healing scripts are essential when the underlying system design changes. Here, autonomous bots automate regression testing cycles, thereby enhancing efficiency. The accumulated evidence from adaptive locators, self-healing, and autonomous bots in regression testing highlights their significant benefits across various scenarios.

4. Autonomous Bots in Regression Testing

A successful DevOps Pipeline helps DevOps teams go faster, but sometimes the integration process slows them down on the journey toward Continuous Integration and Continuous Delivery [6,29-31]. Typically, the integration process must synchronize the entire software's functional and non-functional requirements to address business requirements fully and satisfactorily. An integral part of this validation process, which takes a significant amount of time, is the execution of the testing aim. Hence, the concept of an Autonomous Bot was invented to carry out regression tests.

DevOps pipelines utilize object repositories that store element locators and descriptive details of the test script to perform automation script execution. The Autonomous Bot is a self-healing bot created using AI and ML algorithms to interface with the object repository. It triggers an event to scan the entire application automatically whenever the DevOps pipeline fails during script execution. The bot compares the differences in the application and the stored script element locator, analyzes the changes needed in the regression test script, and updates the object repository accordingly.

4.1. Role of Autonomous Bots

Autonomous bots in test automation streamline processes, minimizing human intervention [32,33]. They independently identify and address errors, debugging and carrying out self-healing as required. AI bots can initiate a test cycle upon the completion of server deployment, eliminating the need for manual triggering. Bots designed for chatbot testing autonomously navigate and conduct tests on chatbot interfaces, while others initiate integration cycles for different teams. Automated status monitoring bots keep track of test progress and dispatch customized alerts, updating key stakeholders. The self-healing intelligence enables adaptive test automation by proactively uncovering and rectifying errors during execution, greatly reducing the maintenance burden for automation teams.

The Evolution of Self-Healing in Test Automation Self-healing test automation stands as a pivotal AI-driven advancement that aids bots in autonomously detecting and remedying failures. Achieving complete bot autonomy in code maintenance substantially diminishes dependencies and underpins continuous integration and continuous deployment programs. The self-healing mechanism is activated when any test step deviates from its intended path. These failures are immediately detected, and corresponding requests—either for new locator suggestions or for web element revalidation—are initiated. Such provisions facilitate the updating of locator configurations for affected test steps, thereby enabling the test automation cycle to continue uninterrupted and successfully complete.

4.2. Benefits of Using Autonomous Bots

Another proposed idea is to introduce Autonomous AI Bots to maintain optimal execution for test suites in Regression testing [34-36]. Regression testing is an extremely important phase in the SDLC (Software development life cycle) process and also very labor-intensive. It requires a huge investment of both capital and human resources on an ongoing basis. Moreover, the testers get frustrated during repetitive regression testing and often do not maintain their optimal focus during regression testing execution and are prone to missing critical test failure results that might even cause the product to go down in production.

Autonomous bots can run any test cycle smoothly, consistently, efficiently, and reliably without getting bored and irritated by repetitive tasks. By using these innovative ideas, organizations can save up to 30% of their total test capital investment. Manual testers can shift their focus on more value-added activities. The product stability will improve, which will have a direct positive impact on the customer satisfaction index.

4.3. Challenges and Limitations

It is important to recognized that AI is not a silver bullet—overstated by too many. AI is the hottest technology topic of the day and is in danger of being misapplied [16,37-40]. Ultimately, AI is a tool—a sophisticated tool that mimics human edit semantic and cognitive processes. The intelligent algorithms do not replace humans but instead work alongside engineers and testers—enabling, empowering, assisting, and improving human capabilities.

Smart content addresses user queries contextual to specific audiences. Web sites and apps deliver both static content and dynamic replies tailored to the real-time characteristics of the user such as demographics, behavior, and supported device. The resulting user experience is better—and the user engagement is higher. Yet many web sites still hand-code these experiences. They require expensive onsite IT staff to deliver simple variations for attributes users, roles, and languages. Furthermore, changes typically require additional reviews and approvals—adding to implementation delays, costs, and other risks. It is also not limited to test automation but used in security, development, audit, and compliance.

5. Integrating AI in Test Automation Frameworks

The integration of AI into test automation frameworks employs AI algorithms to enhance the analysis capabilities of feature files, the automatic generation of scripts, and the maintenance of existing scripts. AI-driven tools enable the development of self-healing systems that minimize the need for human intervention within the automation cycle. These systems utilize calculated optimal paths to perform necessary sequences of actions, improving the resilience and adaptability of test automation.

These capabilities result in significant reductions in business maintenance effort by decreasing the inflow of missing element bugs from the test environment. The application of AI thus enhances not only the efficiency of the automation processes but also the overall quality of software products by automating maintenance tasks and enabling new functionalities. The self-healing algorithm represents one such newly developed feature.

5.1. Framework Design Considerations

Test Automation Framework Design — Recent research articles claim AI/ML tools and algorithms can be used to design “self-healing test automation frameworks.” However, little has been found based on design principles for such frameworks. The general requirements for a test automation framework are:

1. Execution planning: Controlling test execution. For example, Login tests must be executed before creating a new account. 2. Test data management: Injecting test data in the framework. For example, when creating a new account, the username must be injected from an external source. 3. Page Object Model: Separating locators from business logic. 4. Logging and report generation: Logging the test execution details and generating reports. 5. Test result validation: Validating the test result with the expected result. For example, after clicking the add button for the new account, the page must validate the presence of the new account. 6. Maintenance: Easy creation and updating of tests.

5.2. Tools and Technologies

Self-healing test automation dynamically recovers from unexpected failures during execution, in contrast to static recovery solutions that allow only predefined exceptions. Various techniques are suitable for implementing self-healing, and numerous open-source tools are recommended for their integration. AI-driven tools related to self-healing in test automation are reviewed and analyzed.

SAP Intelligent Robotic Process Automation supports both the design and operational phases through Task Bots and Process Bots. Google's Cloud AI offers models for features such as Language Translation, Speech-to-Text, and Text-to-Speech. Appvance IQ employs AI techniques for test generation, action learning, root cause analysis, and self-healing activities. Mabl utilizes machine learning to automate test design and enable self-healing. Testsigma applies AI and NLP for intelligent execution, spotting pattern failures, and self-healing. Functionize uses natural language processing, machine learning, and computer vision. ACCELQ performs risk-based test automation with machine reasoning. CA Autonomation leverages machine intelligence for healing and automating test activities. Genpact employs machine learning and natural language understanding to create self-healing test scripts and analysis reports.

6. Real-World Applications of AI in Testing

AI-supported systems possess an abundant nature of environmental information and can actively adapt to changes in their environment to optimize their own performance [7-9]. These capabilities are critical during the testing phase. If software testing can also be intelligent, it will enable automated creation, detection, and repair of test cases, thereby reducing dependence on individuals.

At present, the integration of AI within smart testing remains a limited area and is reflected primarily in methods, frameworks, and techniques. Clients typically need to

submit defects or requirements, and AI then tests accordingly. However, the industry still lacks fully integrated intelligent testing frameworks. Real-world test automation projects, which employ AI solutions for self-healing and self-repair in the testing domain, demonstrate that AI is progressing and will inevitably become an integral part of testing.

6.1. Case Study: Company A

In the software test industry, test execution and its reporting consume significant time and manual effort. These processes are vulnerable to human error, leading to a considerable number of defective executables in the production environment. Utilizing AI in test execution can reduce manual effort and associated errors.

Company A is an ecommerce platform with an integrated payment gateway. All transactions are linked with credit cards from six different banks. Company A experienced numerous production bugs when the payment gateway was connected. They deployed test automation scripts in production to validate each transaction. One run requires 1,700 tests, taking around 10 hours to execute. The executable reports must be quality-checked before sharing with the client; each report is approximately 2,000 pages, a task that is time-consuming and tedious. The company implemented an AI solution to automate this process, significantly reducing the time needed for test execution and report preparation.

6.2. Case Study: Company B

A practical application of AI in software testing can be illustrated by Company B, which turned to AI to develop a self-healing tool. Technical leads at Company B acknowledged that the tool's AI features were in their infancy, yet nevertheless valuable. The initial challenge was to design, develop, and deploy a simple, browser-agnostic, and locators-centric test framework for web applications. The first artificial intelligence feature added was a Smart XPath Locator — an AI locator capable of predicting the most relevant XPath for a particular element. The second self-healing feature was applied to the test-execution engine. The engine monitors the occurrence of element locators in executed test case steps, and when a locator fails, the engine recommends alternative XPaths or utilizes the Smart XPath locator to suggest alternatives. These alternatives can then be automated for self-healing and recommended to testers within the tool. The third feature employed an object detection model based on AI, enabling the detection of page components and increasing the robustness of test cases when they need to be changed due to UI changes.

6.3. Lessons Learned from Implementations

What has been learned from the implementations of intelligent test automation and self-healing? Several points repeat those already discussed. One fundamental finding is that AI-based test automation is not necessarily easier, as one vendor claims. In fact, some scanned presentations took a simplistic approach and missed several key points that had been made earlier in the first category. Other presentations did acknowledge the difficulties involved with skilled, say, test automation developers. An example is the Intelligent Schedule that is not possible without person skills and his/her hands on experience and his/her knowledge and understanding. A second key lesson is that many of the drivers remain untouched and unchallenged. For example: “Test automation tools that require less effort in script maintenance would be highly desired.” or “Today, companies struggle with resource-limitation issues and perform regression testing missing major test cases, resulting in low test effectiveness and product quality.” and “Modern test automation tools require satisfactory maintenance efforts with enhanced capacity with the use of AI.” These comments seem to confirm the issues raised in the first category.

Perhaps the most important point is that the four reasons for failure have not yet been refuted or eliminated. The outcomes have to be measured in real cases to see whether the weaknesses of data and model may have been suppressed, that resource limitation disappears, and that more AI capability reduces test suite design and resource effort. Addressing the failure reasons and the areas remaining important for self-healing: test execution, test script maintenance, and test suite design—as well as enhancements to schedule generation—are then the most promising directions.

7. Future Trends in AI-Driven Test Automation

Test Suspending, Saving and Resuming If the billing application undergoes a lengthy regression test with hundreds of tests and an allegation is raised against an environment related to the build, the entire test suite needs to be aborted. This means that the entire testing cycle needs to be started once again when the environment issue is resolved. By saving and resuming the test execution, the test suite may be just paused during the execution until the environment gets fixed. In this case, a developer or a test automation engineer needs to suspend the process, save the execution logs, loading history and the system status, and finally be able to resume the execution post fixing the environment. By offering a solution to suspend a test, save its current state and resume the test execution after a failure takes place, continuous testing across any software development life cycle stage may be enabled.

Test Prioritization Test prioritization involves ordering the way in which tests are executed. Some tests are more important than others and so identifying and running the most important tests first, ensures testing uses less time, is more effective and has maximum risk coverage. Tests can be prioritized based on different criteria such as: Importance of the functionality under test, Importance of the testing, Functionalities that are being frequently used, Functionalities that are being more prone to defect finding, Test stabilization issues.

Test selection helps optimize the testing effort by selecting and running only a subset of the existing tests. Test selection might be required at various points in time: After a defect is resolved to verify the fix. After a functionality is completed to verify system stability. To enable faster completion of regression testing in CI and nightly builds. Selection criteria can be: impacted components, users, environments or customer journeys. It can be based on the coverage of different scenarios or recent execution results. The selected tests should cover all the above factors in an optimal manner.

7.1. Emerging Technologies

AI holds significant promise for software testing, and the broad trend of applying it to IT operations, known as AIOps, is already benefiting test groups. AIOps offers early patterns useful for steering test groups to both elevate their performance and achieve Self-Healing. The maturity of AIOps is shown through vendors such as Moogsoft, Splunk, and BigPanda, providing machine learning solutions that distinguish informative alerts from noise, thus assisting shift-left test groups in troubleshooting test or application failures.

The current AI-based test tools market is fragmented. Although test tool vendors are eager to demonstrate AI capabilities, they lack clear market guidelines. The focus up to now has been on supporting execution and defect analyses. However, early AIOps successes around noise elimination are beneficial to test groups and, when combined with a shift-left Agile mindset and testing crowds, pave the way for Self-Healing.

7.2. Predictions for the Next Decade

In the section "" the future of AI in test automation is presented. AI is expected to improve smart agent capabilities and the smartness of connectors. Data-driven testing and autonomous testing might be triggered by self-healing test automation, which is AI-driven. A working model for self-healing test suites, a system that automatically detects errors in test cases and fixes them, is outlined. Such a model would play a crucial role in reducing human intervention when errors are detected in the test suite. Self-healing in

test automation is suggested as a partial replacement of human testers in error detection and resolution. During test suite execution, if execution is unsuccessful, the self-healing model should automatically detect the repair possibilities and present them to the tester. Based on AI, such a module might even heal the tests automatically, without human intervention.

The lack of end-to-end coverage will continue to be a challenge. Testing at the business level will become more data-and-failure-driven instead of code-structure-driven. There is significant potential for code-level and unit testing by smart agents and algorithms. Agile and DevOps will continue to grow, and a growing population of nontechnical users will use these technologies. At the business level, there will be increased reliance on test execution coverage supported by data-level triggers. Smart test-execution algorithms will determine the subset of tests to be executed, and they will select the set of test data for each selected test case. Current UI-based automation has limitations when customer expectations change midway through the sprint cycle. Modifications to the UI can be easily handled by the self-healing module. Despite that, creating the actual tests can still be tedious. Future-generation test automation technologies need to become more ‘‘clickless,’’ supporting nontechnical users with natural-language-based test authoring and execution.

8. Conclusion

AI has been implemented in test automation. Intelligent methods are not yet mainstream in commercial test automation tools; however, the field is evolving. Most existing open-source and commercial test automation tools rely on image-based properties using a coordinate-based approach that requires constant maintenance. Even AI-powered commercial test automation tools that use ML for recognizing application objects depend on computer vision and image-based properties. Furthermore, evolution-based methods focusing solely on expected output data may prove helpful for unconditional code coverage testing. In reality, test automation is excessively brittle, resulting in huge maintenance overheads.

Test automation combined with AI empowers self-healing capabilities. Test automation scripts can be made self-healing by performing bidirectional mappings between GUI-based low code elements or image-based output and code's expected values using AI-powered OCR, NLP, and ML techniques. A self-healing test automation engine can be architected by combining a coverage-based test data generator with an execution engine. The engine generates automated test scripts using coverage-based test data generated by parsing source code, thus removing the dependency on external files such as Excel sheets. During execution, AI applies statistical methods and OCR techniques on the actual GUI output to extract facts, which are then resolved with expected assertions

present in the test code. By analyzing the difference between expected and actual assertions, AI adds new conditional branches in the test method to handle different actual outcomes in future execution cycles, enabling fully conditional code coverage-based testing with an image-based approach for expected data handling and self-healing features.

References

[1] Bari MS, Sarkar A, Islam SM. AI-augmented self-healing automation frameworks: Revolutionizing QA testing with adaptive and resilient automation. AIJMR-Advanced International Journal of Multidisciplinary Research. 2024 Dec 1;2(6).

[2] Baqar M, Khanda R, Naqvi S. Self-Healing Software Systems: Lessons from Nature, Powered by AI. arXiv preprint arXiv:2504.20093. 2025 Apr 25.

[3] Pandhare HV. Future of Software Test Automation Using AI/ML. International Journal Of Engineering And Computer Science. 2025 May;13(05).

[4] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.

[5] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.

[6] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.

[7] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.

[8] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.

[9] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.

[10] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.

[11] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.

[12] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.

[13] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.

[14] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE

International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.

[15] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.

[16] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.

[17] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.

[18] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.

[19] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.

[20] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.

[21] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.

[22] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.

[23] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.

[24] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.

[25] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.

[26] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.

[27] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.

[28] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.

[29] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.

[30] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.

[31] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence

technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.

[32] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.

[33] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.

[34] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.

[35] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).

[36] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.

[37] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.

[38] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.

[39] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.

[40] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 7: Natural Language Processing in Software Quality Assurance and Testing

Partha Mohapatra

AT&T Corporation

1. Introduction

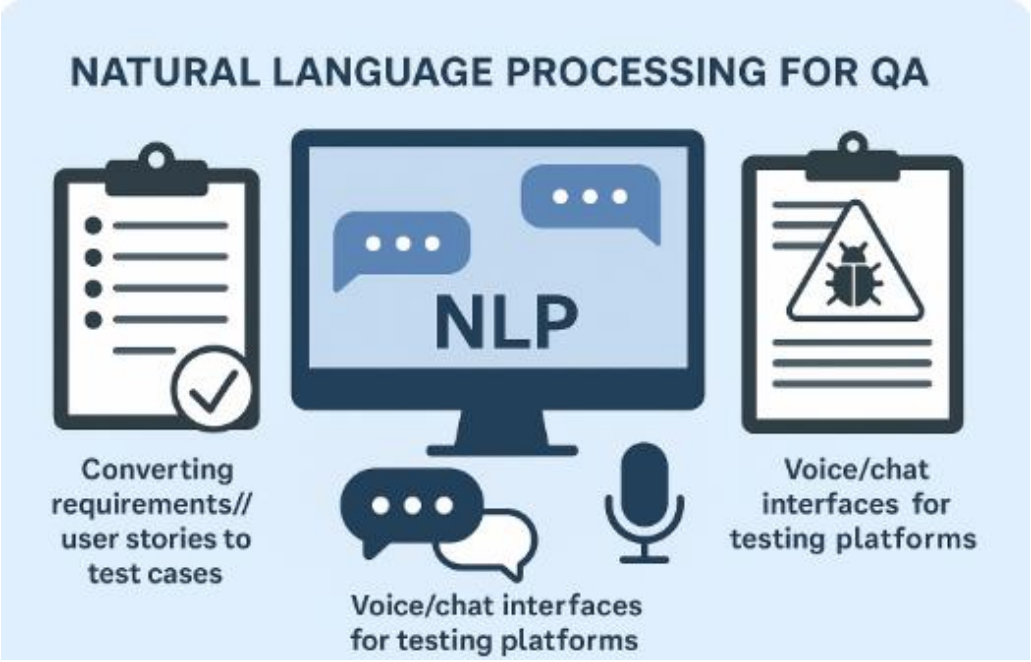
Inspection comprises reviewing quality assurance plans and practices, overseeing test activities, and delivering the product to the customer [1-2]. The overarching objective of inspection is to increase the probability that the product adheres to the relevant quality requirements. Testing requirements may derive from the SRS document or other sources. Testing generally involves exercising the software using automated test scripts, manual tests, and other approaches [3-5]. The requirement verification process provides product metrics related to test cases, such as the number of passed tests, blocked tests, defect distribution, and severity levels. Acceptance criteria can include the pass report of acceptance test cases indicated by the Requirement Verification team; stability metrics, such as the number of regression bugs in a period; and customer confirmation.

2. Overview of Natural Language Processing

Natural language processing (NLP) is a demanding area of artificial intelligence that offers tools and theories dedicated to the analysis and synthesis of natural language and speech [6-8]. Furthermore, NLP is one of the most complex areas of computer science and artificial intelligence that studies how to program computers to process and analyze large natural language-materials. The central goal is a computer capable of "understanding" the contents of documents, including the contextual nuances of the language within them.

One of the many fields where natural language processing can be used is quality assurance (QA) in manufacturing. The goal is to design and develop a domain-specific

ontology and to retrieve the information in free-text documents. At the conceptual level, the challenge is to model specific natural language knowledge effectively that is reusable and extensible for the domain.



3. Role of NLP in Quality Assurance

Natural language processing (NLP) has changed the way software testing is done. It offers vital benefits for test automation, creating sentences in user-friendly language that offer clear instructions to the auto-generated tests that can be applied throughout the testing lifecycle [7,9-10]. Specifically, it ensures test scenarios are created quickly, accurately, and completely written using natural language.

The testing process is set up through a fully integrated process that extends the business requirements [1,11-14]. These steps include testing process initiation, test execution, evaluation, and approval. Once a set of test scenarios is downloaded, testers can revert to the initiation step by proposing any desired changes [13,15-17]. Changes are evaluated and, if accepted, returned to the approval and implementation phases of the application. The NLP approach also makes it easier for users to enter regression test scenarios into natural language. When the user selects a particular scenario and clicks the Execute Test Case button, the program builds the vUser scripts code based on the natural language test scenarios.

4. Converting Requirements to Test Cases

The practical application of the described approach proceeds from the requirements specification in natural language to the generation of system test cases also in natural language. An example newly written for plastic bags prevents the discussion from being dominated by a recurring example of an ATM system and clearly shows the domain independence of the approach. The final report of the scope is abbreviated and reduced to the aspects analyzed.

Whenever the meaning of a requirement has become clear, the related cycle of generating objectives and test cases starts [18-20]. The stages of the approach as shown in Figure 1 are treated at the same time and in a dynamic sequence. In practice, each additional objective or test reveals a gap in the scope, test or support set, and the text in the cycle still needs to be completed. The approach is illustrated by the first test case generated from the example.

The objective of the support vector is to ensure the customer support priority of CAT against the policy given all the details of the objective that tests this priority that are generated below. It expresses the expectation that a support request is closed within 8 days. It is formulated in the beginning of the text because it still expresses the intention of leadership that was the background for the policy.

4.1. Understanding Requirements/User Stories

Much of the technical information that describes a software product is written in natural language. The use of natural language in product descriptions also affects quality assurance (QA) [19,21-22]. Specific tasks in QA, such as testing and reviewing of natural language written documents, are known to be cumbersome and error-prone. In agile development, the user stories are the sources used by developers, testers, and other project participants to understand the features that the system under development is expected to implement. Imperfectly written user stories can lead to many negative impacts on the software project, for instance, in the form of wrong implementations, bugs, or even project failure. The application of Natural Language Processing (NLP) in software engineering is emerging as an interesting topic, enabling a variety of services in the high-demand area of automation.

The way how requirements and user stories are written represent constraints for those users' stories to be used [11,23-25]. It is well-known in software engineering that wrong requirements can have fatal consequences for software projects. Worse, those requirements are usually written, at least initially, in natural language (NL). But NL is naturally ambiguous and writing good requirements is not an easy task. Testing is the major activity to verify the software system. However, it begins with the understanding

of the product. Testing cannot be done without product knowledge and product information understanding. A software product and its features can be described in multiple ways amongst which the test-base documents are the user requirements. User requirements have significantly evolved. These have moved to Agile methods.

4.2. Techniques for Requirement Analysis

Requirements are acquired from the client in a client meeting and are then analyzed and written in the requirements document, which is then passed on to the client for approval. After approval, the development of the project begins [26-28]. The requirements document consists of the whole information needed for the project, which includes the algorithm of the problem, the input and output requirement, the interfaces to be used, the structure of the project, the deadline for the project, anything that has to be done in the development of the project. NLP techniques can be used to analyze these requirements documents and check for uniqueness, ambiguity, and correctness.

Requirements Analysis, also known as Requirements Engineering, is a fundamental step in the project development life cycle. It involves understanding the services that customers require from a software system, the constraints under which it operates, and the goals an organization hopes to achieve through its implementation. In fact, a substantial portion of every project's budget and effort must be devoted to this activity. Requirement Analysis aims to define the organization's requirement as precisely as possible so that everyone involved has a clear understanding of the business needs and objectives being addressed by the software system.

4.3. Mapping Requirements to Test Scenarios

The Value of NLP in QA Using NLP to analyze requirements can complement the expertise of seasoned QA professionals. NLP algorithms do not suffer from tunnel vision or preconceived assumptions about system requirements, business model, intended functions or underlying coding. Instead, they rely exclusively on cues that can be identified in written documentation. Although NLP models lack common sense and real-world knowledge possessed by most people, they do have the ability to parse and understand large numbers of requirement documents. This capability can both improve the completeness of mapping requirements to test scenarios and reduce the time and resources required.

Mapping Requirements to Test Scenarios Generating a detailed map between test scenarios and business requirements provides two main benefits. First, analyzing the mapping helps guarantee that all requirements are accurately verified through the testing

process. Second, the map creates a reference document for developers and business analysts that facilitates updating requirements test scenarios during the changed requirements or regression testing phases. Typically, mapping requirements to test scenarios is a manual process. Although it can sometimes be partially automated by analyzing memoranda, related bugs and other artifacts, a manual approach is regarded as the most reliable way to establish the mapping accurately.

4.4. Case Studies and Examples

The use of natural language processing for inspections and quality assurance is not a new idea; accidental findings, quiet reporting, social media, customer support, and changes in regulations are all potential sources. These can be parsed as unstructured text sources, possibly with additional metadata like region or industry. Source types include: Inspections and quality assurance documents or a specialized process for expert assessment.

A particularly extensive involuntary reporting source is the datasets that have been collected for product recalls. A natural language pipeline can be used to extract specifications, often as part of a quality management cleanup project. Examples of NLP in inspection and quality assurance include:

• FDA Recall Compliance Pipeline: Recalls are an important factor for the overall quality and safety of food, drugs, cosmetics, and other products. The U.S. Food and Drug Administration (FDA) publishes recall specifications as semi-structured text that must be compiled from multiple sources, including the FDA and other agencies. The unstructured recall description is mapped to a structured form for further analysis, such as risk assessment or cause-effect analysis of recalls.

• Automated Product Breakdown Structure Pipeline: In some cases, the product breakdown structure is not available in a consistent and reliable format. It can be extracted from semi-structured documents. Combining available structured data with new product structure data automatically extracted from legacy documents will feature possible applications in change impact analysis, risk assessment, root cause analysis, and others.

5. NLP-Based Bug Report Analysis

Producing a test case for all possible scenarios involves vast amounts of work and undesired costs, making it unfeasible to satisfy all test conditions [29-32]. Therefore, analysis methods decide the test conditions to be evaluated in the test case prioritization phase. Prioritized test conditions focus the test case on critical categories in software;

for example, the fatal category could be tested first. One approach that can efficiently analyze and categorize bug reports is natural language processing (NLP).

According to Sornam, NLP is the automatic manipulation of natural language, like English, by software. Drawing from the spice identification analogy, where the software cleans and processes spice into a new avaible format, NLP allows software to read and understand human languages. Bug reports, often structured like emails, resemble natural human language. In addition to analyzing the priority of different test scenarios, NLP methods can parse, summarize, and summarize bug scenarios.

5.1. Importance of Bug Report Analysis

Quality Assurance (QA) and Testing represent two of the most significant consumers of the resources in the entire Software Development Life Cycle [31,33-35]. For most projects, they are also the most prone to late stage changes due to the evolving perception of the business environment. Thus, developers are often called upon to introduce a change in the implementation of a particular module or several different modules to enhance the capabilities of the software. However, owing to the numerous constraints like budget, time or resource availability, many of the changes may be implemented/merely planned and not fully tested before the release of the software. This is particularly true in waterfall-model based projects, where testing often begins very late.

The developers who have implemented the change(s) or have been made aware of it, usually inform the QA-testers about the changes so that they can test its impact on the other functionalities. However, without a semi-formal notation for requirements or changes, the testers may either not understand the implications fully or may not know of the specific modules to test against them. Furthermore, in an agile environment, the tester is often given a list of new features so that the modules initiated to implement those features can be tested properly. The tester is unable to find out the list of other features which were tested but which are likely to get impacted by the new code, since there is no mechanism to prioritize modules affected by the recent change. In either case, an important task is to allocate the tester to the right module such that the probability of the artifacts being rejected by the business user is minimized.

A naive prioritization technique would involve the prospective tester understanding the entire business, the software module(s), and the associated user cases. These would then be mapped to test cases after several iterations in order to assign the next test cycle. However, the other alternative (and the one recommended in this paper) is to analyze the already existing bug database for the software product which is under test. Using this process, the tester need not have knowledge of the application feature, rather is provided

with an overview of the defect types that appear on each module. Information from the earlier bug reports can help prioritize future testing and also help allocate the right tester to the right module. Further, the priority of that module testing is automatically derived depending on the projected severity of that test cycle.

5.2. NLP Techniques for Bug Report Processing

The manner in which a bug report is written also has a direct impact on the accuracy of duplicate bug report detection. Customers must be instructed to present the bug report in such a way that the information in the bug report can be comprehendible by a computer—English in the case of Bugzie—and the tasks necessary to resolve the bug report can be predicted. It may even be necessary to implement bug report suggestion during writing and completing. As the number of bug reports increases in this digitalized world, considering consumer convenience is essential.

Evidence exists for the use of NLP to process bug reports. NLP extracts information from bug reports and then uses that information to execute the desired tasks. The use of NLP techniques with the different parts of the bug report is necessary. Often, only the title or only the description is used, but in some studies, the title and description are combined. The problem title and description influence the prediction of the time to the first decision; the category of the bug is predicted based on the problem title; the total time to resolve a bug report is predicted based on the description; and the responsible team for solving a particular category of bug is predicted from the description.

5.3. Sentiment Analysis in Bug Reports

Sentiment analysis or opinion mining in software engineering involves analyzing natural language expressions in various software development artifacts to determine the emotional tone behind the words [36-37]. Sentiment analysis of bug reports in natural language is primarily applied during the quality assurance phase of the software development life cycle using linguistic constructs such as Manually Annotated Releases, Text, Developer, and Product, commonly abbreviated as MADIT.

5.4. Automating Bug Triage with NLP

TF-IDF is the main vector space calculation. It matches tokens from the processed description to tokens in the description or in the fix that was made to the code repository. NLP seeks to automate the assignment of described bugs to developers, linking them to specific areas of code. In the initial stage, TF-IDF algorithms frequently examine the

fields of the bug and search for similar bugs that have been matched. This offers a list of suggestions for developers who might be related to addressing the bug. The vector space model utilizes two fields for this calculation: the description, what the bug is about, and the component, the part of the program where the fault is located.

TF-IDF measures the weight of each term or token in a document when searching for similar documents. The importance of a term increases based on its frequency within the document but decreases with its commonality across the corpus. The algorithm employing TF-IDF operates in two phases: the training stage and the prediction stage. During training, the corpus undergoes preprocessing to remove HTML tags, punctuation, stop words, and building tokens. After training, new bug reports are compared against the corpus to generate a list of ranked bugs. By leveraging the similarity between descriptions, the algorithm proposes developers who might be suitable for handling the bug.

5.5. Challenges in Bug Report Analysis

During the analysis of bug reports, a number of complications can arise. Any potential defect must be reproduced before a resolution can be created, which can be difficult because the description of the defect may be unclear or erroneous. Even when the defect can be reproduced, it may take a significant amount of time for it to be fixed. Furthermore, bug resolution generally requires developers to gain an understanding of the defect and its causes, both of which are generally not included in the bug reports themselves. Finally, not all bug reports are associated with a real defect. Bug reports that do not address a legitimate defect are a major cause of wasted maintenance effort.

Being able to automatically analyze bug reports to verify the accuracy of a report, reproduce the defect, or identify potential resolutions would greatly assist software maintenance and improve software quality. However, performing such analyses is difficult because bug reports generally contain unstructured natural language text, making it more challenging to identify the important terms.

6. Voice and Chat Interfaces for Testing Platforms

Research in voice and conversational agents has increased substantially over the last years, mainly boosted by Alexa, Google Home, and Apple Siri for billions of users worldwide. Venturing into the voice and chat interface rapidly opens up new markets and industries, such as conversational commerce, where conversational agents actively sell and cross-sell products. Other areas, including banking or general customer support, aim to steer users toward specific webpages or actions instead of sending them an online

form or requiring a call. Further applications include voice dictation and voice content creation.

Integrating voice and chat interfaces with Salesforce requires new testing strategies. Functionalities that were previously tested using predefined flows must now be tested with the diverse ways users may ask for the same thing, leveraging natural language processing. The testing of the Salesforce platform must transition from count-based tests to natural language-based tests. Ironically, testing agents that simulate human conversations involve the application of natural language processing—historically a notoriously difficult task for computers.

6.1. Introduction to Voice Interfaces

Speech control systems enable voice-driven operation of machines, potentially without supplementary devices or training phases, reflecting a direct mapping between linguistic commands and control or query parameters. Regardless of the recognition system's internal representation or the language model's grammatical framework and command semantics, a common structure is often imposed on the spoken command set, which is typically predefined. In industry, tasks involving a high share of communication in standard language can be efficiently supported by voice control, utilizing specific domain knowledge. For example, requests for information or simple command sequences that cannot be expressed using standard control vocabulary could be formulated in natural language.

Such a system not only enables simple allocation of individual seat requests but also guides the user through the request procedure. The core of the system comprises a Voice XML Control Server, which communicates with the corresponding module within the Voice Response System, also known as the Dialog Manager. This system module analyzes both the users' speech input and the internal database feedback to generate instructions for the text-to-speech engine. It also controls the Voice XML Server, assigning a specific voice profile to the synthesized text output based on the users' demographic variables. The synthetically generated speech outputs form the system's answers to the requests defined by the users' speech commands.

6.2. Chatbot Applications in Testing

Chatbots can be taught to interrogate systems, execute use cases, and verify system behavior. Basic chatbot applications in testing require some form of dialog-based interaction to control the SUT bus. The dialog function must accept inquiries, execute required routines on the SUT bus, and subsequently relate the results of those routines

back to the investigator in the dialog environment. When the investigation is a question aimed at executing test input, the dialog function must accept the investigator's inquiry, execute the required inputs on the SUT bus, and report the results of the execution back to the investigator within the dialog environment. The natural language form guides and controls the SUT bus implementation and verifies that the proper tests are selected and executed.

Advanced natural-language recognition and interpretation modules also facilitate the hopeful-step requirement. Whenever a suspicious test result is recognized, the chatbot analysis module is triggered. The module then requests the successful-path requirements from a natural-language requirements database, performs a temporary comparison of the suspect-case steps to the successful-path-case steps, and generates a series of dialog-based questions and statements seeking reconciliation of the suspect case to the successful path.

6.3. User Experience Considerations

Performance considerations are not unique to natural language processing models. They arise whenever a user interacts with a system in a way that makes the user appear to be waiting for the system to respond. When a user experiences that kind of delay, psychological studies have identified three breakpoints in relation to the extent of the delay.

The first break, approximately 0.1 seconds, is the point at which a user notices that a system is responding to their actions. The second break, on the order of 1 second, is the point at which a user notices that a delay is occurring, but the response does not interrupt their thought process. The third break, about 10 seconds, is the point at which a delay is sufficiently long that the user abandons the task altogether.

6.4. Integration with Existing Testing Tools

Integrating NLP into existing test automation frameworks allows a hybrid strategy that combines unstructured, human-oriented test cases with conventional, highly structured ones. This integration makes an immediate impact on testing teams. Bots are trained with traditional test cases, then executed in parallel with existing tests. The automation backlog is gradually replaced with test cases written in natural language, and the existing manual backlog is systematically reduced through reusability.

Complex platform settings challenge test automation tools and bots. The higher the level of platform integration, the more complex it becomes to set up for testing. Executing a test case often requires setting up the system with specific data and configurations.

Moreover, cross-application policies demand the evaluation of common system criteria, such as GDPR compliance, prior to further test case execution. For instance, in web development, WebDriver manages full browser execution.

6.5. Future Trends in Voice and Chat Interfaces

Voice and chat interfaces have made great strides for speech recognition, enabling hands-free conversational information access, web search, weather forecast checking, and other applications. Research has uncovered significant potential in healthcare, especially in multimodal applications that synchronize imaging with verbal explanations, speech synthesis applied to echocardiography, or embodied conversational agents in coaching roles. Investments now focus on creating interfaces capable of executing complex tasks such as checking the weather, searching online content, or adjusting a thermostat in a smart home environment.

Recent efforts introduce a novel chatbot concept driven by deep learning, featuring three main components. The first determines the topic of a user's request, guiding the selection of a reply search space. The second generates the actual reply, based on the user's request and the unique information model of the project. The third validates the accuracy of the reply.

In response to these requirements, it is proposed to segregate natural language processing into context-oriented capabilities and language-oriented tasks. The chatbot receives its own language engine to fulfill a broad range of linguistic functions, independent of any specific context, while a separate domain engine combines domain knowledge with the language itself. Together, they ensure the production of timely, correct, and consistent answers.

7. Evaluation of NLP Techniques in QA

In quality assurance, natural language processing can be applied to build a system that automatically checks code comments and crowdfunding campaign descriptions. The Checkbot Comment-Codes system determines if comments are related to the code they describe. Using word embedding and the classical machine learning method of Support Vector Machines (SVM), the system represents comments and code with vectors and calculates their similarity. Two publicly available datasets are labeled for related-ness following a three-step strategy and used for evaluating the approach. The results demonstrate better performance than random-selection and trivial baselines.

FinTextCrowd assumes that the quality of a campaign description is related to the success of the funding project on crowdfunding platforms. The study tries to calculate

the quality of the description by using quality-related features from readability, linguistics, and syntax. Regression analysis determines the relationship between the features and the success rate. The best-performing sets of features are adopted to build an NLP approach that scores the quality of a given description. Evaluations show that the predicted scores and the real scores of the descriptions are strongly correlated, indicating that the quality scores can measure the likeliness of a successful campaign and support the improvement of the description.

7.1. Metrics for Evaluation

Prompt engineering can be optimized on top of the general LLM model using specifically designed metrics. The metric defines a so-called reward function giving the degree of correctness for a generated answer. It could be as simple as a hard step function with a pre-defined threshold determining whether a generated answer is correct or not. However, a better approach is to have a smooth and fuzzy degree of correctness that gives more rewards for better output and less for worse; it can reduce the likelihood of generated hallucination,response irrelevance, or redundancy.

The existing metrics group into many categories ranging from classical ROUGE and BLEU n-gram overlap measures to model-based evaluation. For example, EASSE is a Python package that makes it very easy to calculate various automatic evaluation metrics for simplification. The large group of automated evaluation metrics of text generation summarizes 17 possible clusters. It proposes BERTScore, MoverScore, and BLEURT as the preferred choices for high-quality text generation evaluation for semantic similarity. T5Text can provide standard GPT-style prompting evaluation for ROUGE, BLEU, METEOR, and BLEURT metrics.

7.2. Comparative Analysis of Techniques

Natural Language Processing (NLP) finds great utility in improving Quality Assurance (QA) operations. Corporate QA uses NLP to evaluate processes, quickly analyze capabilities, and make progress against goals. Broad NLP analysis enables QA executives and managers to understand the most common reasons failures occur. The deeper process root cause analysis identifies the real reasons for failures and enables prioritization of process improvement initiatives. The comparative functioning of some QA techniques is given in tabular form.Section 3 has described the functioning of these techniques in detail.A comparison is made across different dimensions such as plan, human resources (HR), evaluation of HR, test environment readiness, tests executed, tests remaining, types of failures, evaluation of failure, top fail reasons, and process initiatives.

Natural language helps in systematizing QA approaches to achieve greater degrees of repetition and exploit collective learning. It also assists in identifying and exploiting opportunities for product and operational improvement that stem from deeper process-root-cause analysis of errors or failures associated with the sourcing, creation, or operation of a product or service. Finally, it helps determine if QA is doing the right things, doing them efficiently, and doing them well. The implementation of these techniques has enabled a significant digital transformation within the QA organization.

7.3. Case Studies of Successful Implementations

In the domain of quality assurance processes, natural language processing (NLP) demonstrates unmistakable benefits. NLP constitutes the domain of AI dedicated to natural language, concerned with the interactions of computers and any natural (human) language. Through the creation of machine-learning models, it is possible to algorithmically generate natural language or to recognize and interpret it (e.g., speech recognition, text classification). The training of these models is based on human-labeled data. All relevant keywords are assigned to input data elements such as quotations or, in the urban context, car-to-infrastructure messages. Nowadays, comprehensive labeled datasets are offered in many different application domains, such as product reviews, tweet sentiment classification, movie review sentiment classification, news topic classification, weather forecasting, and QA tests.

Few applications draw on the rich manifold of labeled data accessible. The formulation and approximate solution of the QA task is paradigmatic in that it applies a trained model to real-time services that interact with, and respond to, the user-generated queries. However, if small in size and narrowly focused, the scope and significance of the automation is constrained in practice. Classifications at scale with a correspondingly broadened range of supported information needs remain a challenge today. In a settings-specific case, QA may benefit from the support offered by user-generated content, such as forums, complaints, or reviews. Use case-specific repositories with high relevance to QA processes permit supervised and explainable automation. Both the QA requests and the answers are comparable with data from consumer forums and product reviews. Training data can therefore also be obtained here. Such rich datasets have long been available for many application domains. Valuable features can be inferred at high levels of confidence, including the detection of the subject matter of a question without explicit knowledge about the underlying domain.

8. Ethical Considerations in NLP for QA

Ethical considerations in employing Natural Language Processing (NLP) within Quality Assurance (QA) encompass key concerns surrounding data privacy and bias mitigation. The application of NLP techniques often necessitates large volumes of data. It is imperative that the collection and de-identification of such data adhere strictly to privacy standards, thereby preserving ethical integrity. Equally significant is the inherent susceptibility of NLP models and algorithms to societal, cultural, and gender biases, which can skew outcomes and invalidate analytical results. These biases must be proactively and continually addressed to maintain model reliability and fairness.

Sentiment analysis of bug reports illustrates the potential impact of these ethical factors. Beyond sentiment detection, bug report analysis via NLP can identify duplicate reports, classify and assign them, and ultimately prioritize them for resolution. The realized practical benefits of these applications are evidenced in their considerable adoption within real-world bug tracking systems. Yet, without diligent attention to data privacy and bias, such implementations risk undermining user trust and the equitable distribution of QA resources.

8.1. Data Privacy Concerns

NLP is the ability of computers to understand and use natural language. Transforming requirements and user stories into test cases involves first comprehending those requirements through the processes of elicitation, gathering, analysing, documenting, and managing. Activities include creating use cases and user stories, identifying application elements such as actors, business rules, flows, conditions, and expected results, and subsequently mapping these requirements to test scenarios and test cases. Processing bug reports—an invaluable source of information for QA—through diverse NLP techniques is another critical function, with sentiment analysis employed to gauge user satisfaction and streamline bug triaging. Emerging innovations encompass the development of voice and chat interfaces for testing platforms.

The effectiveness of NLP techniques for QA can be examined via suitable metrics, comparative analyses, and case studies, while also considering data privacy issues and potential biases within NLP models. User stories originate from a myriad of settings, including internal team discussions, workshops, meetings, client interactions, and email communications. Input sources extend from testing management tools to Microsoft Office documents, Team Foundation Server (TFS), and JIRA, and are ultimately processed using Scrum and Microsoft Team Foundation Server as the foundational platforms for automating the generation of test scenarios and automated test cases.

8.2. Bias in NLP Models

Machine learning models are data-driven, and their biases depend on the training data. Seeking comprehensive datasets for automatic bug triage remains an open issue because they must maintain project relevance and conflict resolution consistency. Labeling vast numbers of bug reports is expensive and complex, which limits the availability of extensive, labeled datasets. Additionally, replicated bugs are often excluded from datasets, thereby losing valuable information. Large language models trained on open-source program code may pose privacy risks, an aspect that services like Copilot have yet to fully address. Developers should be aware of potential ethical and legal issues when using outputs generated by such models.

9. Future Directions in NLP for Quality Assurance

As the amount of data increases in software development, machine learning is used more frequently in the sourcing, processing and analyzing phases of quality assurance. With the changing nature of NLP, deeper semantic analysis is an area for improvement in NLP models. Semantically comparing test cases with other test cases and user stories with test cases for similarity could reduce inaccurate tests. An endeavor by Google, for example, focuses on text understanding that relies on the semantics of a paragraph rather than on a few keywords alone. The team is currently able to digitize tables and use the information therein as input features into an algorithm.

The application of NLP in the quality assurance process has grown exponentially during the last few years. It is important to consider every aspect of quality assurance, keeping in mind the benefits of the NLP technique. The five elements are sourcing test data, processing it, test execution, analyzing the results and presentation. Although a strong connection exists between NLP and machine learning, each of these elements benefits when they are integrated into the quality assurance process separately. Gartner's report on the state of AI in 2019 indicated that only 1 in 10 organizations deliver AI-based models to achieve desired business outcomes, despite ongoing investments. The problem is rooted in the lack of tangible ROI and also the lack of AI skills. Specifically, the chief data officers recognize the difficulty in moving from the proof of concept to more standardized iterated deployments, even more so when NLP is involved. Thus, more work needs to be done to bridge the gap between the AI industry and quality assurance companies via commercial business processes.

9.1. Emerging Technologies

Other emerging technologies may be worthy of consideration and investigation for use in pQA. Autonomy and conversational AI could help humans to manage increasing amounts of data, and the related risks of information overload. A deep understanding of language and grammar allows technologies based on language rules to be quickly and cheaply developed and updated. Rapid transfer of intelligence and skills to a new domain and environment is possible where the key language features in a particular domain are shared or very similar.

Existing capabilities of automatic summarisation and autocorrect can be leveraged in pQA. Automatic summarisation of information could help to reduce the volume of information requiring manual review, freeing up key personnel to concentrate on other important activities, such as modelling and forecasting. Autocorrect could complement non-autocorrect technology by automatically rewriting the text of detected issues into a more readable form to aid interpretation by the reviewer. Integration could also be considered for other pQA technology such as sentiment analysis, to allow support personnel to rapidly identify events of interest in comments and to extract key issues mentioned in the text.

9.2. Research Opportunities

Q–A Application Support Quality assurance (Q–A) processes related to application support present several challenges. Relying on support for the technical operations of bank applications requires a considerable amount of Q–A. A nonexhaustive list of concerns includes support manuals, timeliness in responding to Critical Incident Reports (CIRs), response quality, incident answering, Service Level Agreement (SLA) compliance, knowledge sharing methods, rework from CIRs and their costs, business impact of CIRs, monitoring of SYS Watchdog, and analysis of business failures, especially those affecting clients directly.

Critical Incident Reports focus on incidents labeled as P0 or P1. P0 incidents are those that impact critical bank clients or processes, causing application outage or business failure. These include failure of transaction manager services, credit card fraud, manual intervention beyond set thresholds, or issues in money-back guarantee processes. P1 incidents refer to system issues related to abnormal server or application behavior, such as high CPU utilization, database latency, and severe deadlock. Errors with a high frequency of occurrence, such as those prevalent in DBSGA, are also taken into consideration.

10. Conclusion

Natural language processing (NLP) allows machines to understand human communication and derive meaning from text. This is particularly beneficial when analyzing text generated for quality assurance of products or services. Using text generated for quality assurance, business leaders can derive key insights that allows them to smoothen their operations. When the text is structured and the categories of groups or classes are predefined, classical algorithms such as naïve Bayes, support vector machines or logistic regression can be used. Creating sets of training data for an unsupervised algorithm can be time-consuming and requires deep domain knowledge. Once the sets of training data have been created, these classical algorithms are less compute intensive and can enable organizations to quickly run the text data through classification.

When the text is unstructured and/or there are unknown probable categories/classes, unsupervised algorithms can group the text naturally. When the text is unstructured and the probable sentiment is also unknown, unsupervised algorithms can highlight those as well. When the text is part structured and part unstructured, a combination of unsupervised and supervised algorithm can be used. This involves using an unsupervised algorithm to group the unstructured text followed by a supervised algorithm to classify the structured portion. Using these techniques, the largely ignored section of unstructured text can be analyzed, thereby delivering better insights that can improve business processes. Quality assurance can be transformed from being a purely reactive measure that is focused on regulatory requirements to a proactive measure that is focused on improving processes and operations.

References

[1] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.

[2] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.

[3] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.

[4] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.

[5] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.
[6] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.
[7] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.
[8] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.
[9] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.
[10] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.
[11] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.
[12] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.
[13] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.
[14] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.
[15] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.
[16] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.
[17] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).
[18] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.
[19] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.
[20] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.
[21] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.

[22] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.
[23] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.
[24] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[25] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.
[26] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[27] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.
[28] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.
[29] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.
[30] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.
[31] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.
[32] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.
[33] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.
[34] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.
[35] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.
[36] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.
[37] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 8: Continuous Testing in DevOps Environments: Integrating Artificial Intelligence for Enhanced Quality

Partha Mohapatra

AT&T Corporation

1. Introduction to Continuous Testing

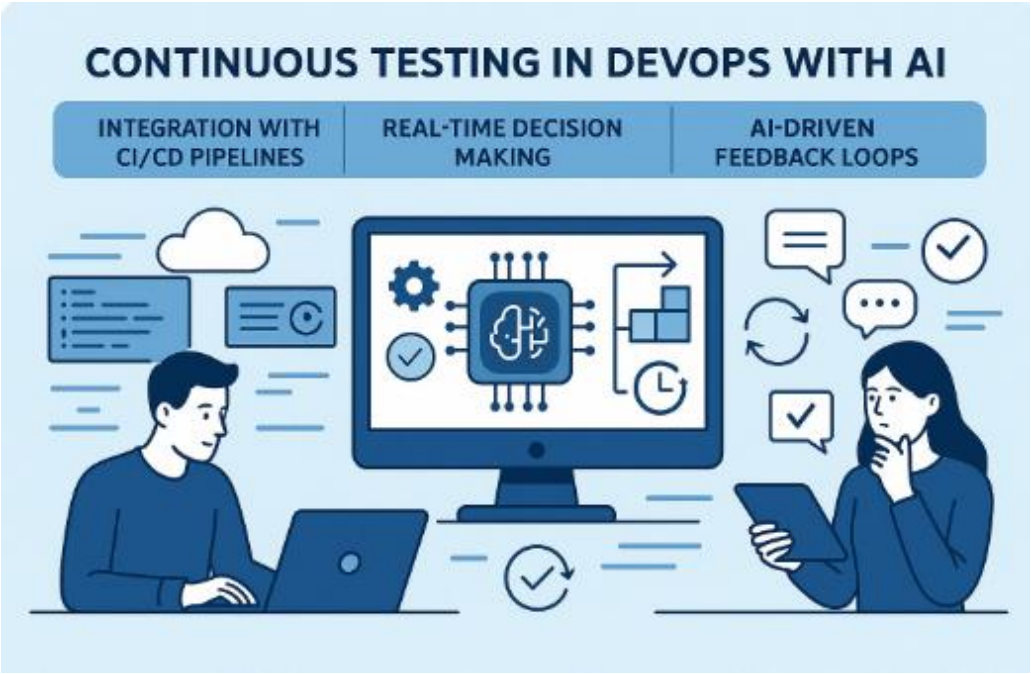
Continuous testing in DevOps is a process that automates the execution of test cases as part of the software delivery pipeline [1-3]. It provides immediate feedback on the business risks associated with a software release candidate, earlier in the cycle, so that risks can be measured and addressed—and the delivery pipeline does not have to wait for the assessment [2,4]. When software is built in a rapid and dynamic DevOps delivery cycle, the testing process becomes a bottleneck. To deliver application updates and code fixes quickly and frequently, organizations are adopting continuous testing to speed the feedback and validation process [5-8]. Continuous testing is integrated with the development and deployment workflows, so testing is triggered earlier in the life cycle, progressively throughout the different continuous integration/continuous delivery (CI/CD) phases, and at a pace that matches development and deployment.

2. The Role of DevOps in Software Development

The software development process is oftentimes performed in a series of stages which include the collection of business requirements, designing and analysis, programming, testing, implementation and maintenance. Business requirements are most commonly collected through communication with stakeholders, and address the expectations the software must meet as well as desired features and use cases. The design and analysis stage involves creating a technical design document in which the software architecture,

features, specifications, requirements and programming schedule are described. These documents are then submitted for approval from different teams of the company such as project and resource planning [6,9].

During the programming phase, software developers write and then compile the source code in the required programming language [10-12]. The testing phase ensures that all features of the software are as specified by the client with every bug identified and resolved before implementation. The software is then released to the area in which it will be utilized. The maintenance stage arrises when a software system, having been installed and accepted, enters the production environment. Within the DevOps system, the release phase of the software development process is performed using an automated deployment methodology.



3. Understanding CI/CD Pipelines

Continuous integration and continuous delivery, as the terms imply, are software development phases in which the build and delivery processes are constantly integrated and deployed. Building denotes development, while deployment is the phase where the software enters production and is released for user use. A CI/CD pipeline is, therefore, a set of practices that allow organizations to deliver code changes more frequently and reliably using automation tools throughout their build, test, and deployment phases.

This approach is used by organizations of all sizes, from startups to tech giants—and nearly universally in software development. It is suitable for any code committed to a version control system and compiled, tested, and deployed using automated tools. Continuous testing is executed at these phases to achieve the desired deployment only if it doesn't break the build. However, when the build approaches release in production, comprehensive testing must cover the entire use period. Continuous testing facilitates test-model creation for all stages of the build by employing AI techniques. These tests are then integrated into the pipeline for execution.

3.1. Overview of CI/CD Processes

Continuous testing is a software testing practice performed during the continuous integration, delivery, or deployment processes in a DevOps approach, where automated tests execute in a real testing environment [7,13-16]. Testing in DevOps is designed to conclude as fast as possible and must deliver immediate feedback to all stakeholders. Making decisions in real time enables organizations to identify problems early in each stage of the software development process. Moreover, continuous testing organizese test automation results into an AI-enabled feedback loop by integrating continuous testing automation tools into a continuous integration/continuous delivery pipeline.

Continuous integration and continuous delivery (CI/CD) represent the next phase of an automated software testing life cycle [2,17-19]. Together with continuous testing and continuous monitoring, CI/CD can reduce the time between writing a line of code and that code being successfully running in production. While continuous integration helps developers regularly merge their code changes into a preproduction branch, which helps identify and fix any issues in the code early on, continuous delivery focuses on automatically pushing software builds through all the stages of the pipeline up to the point of production deployment. Continuous testing allows developers to perform quality checks through every phase of the CI/CD process, thereby preventing bugs from moving further along in the software's lifecycle.

3.2. Benefits of CI/CD in Software Development

Software development teams that implement CI/CD can deploy apps faster and with fewer defects. They also can respond quickly to user feedback and market changes without risking business continuity [3,20-23]. Service-level agreements are easier to meet and maintain because of the reduced risk and the ability to roll back any changes that do cause problems. Analysts have predicted that by 2023, 90% of successful software development teams will have adopted a CI/CD process.Integration of testing into CI/CD pipelines enables development organizations to implement a continuous

testing practice. This approach assesses the risks of a software release at every stage of the delivery pipeline, with AI-driven decision-making determining the timing and scope of testing for each build. Data from multiple integrated tools—such as static code analyzers, dependency vulnerability scanners, and unit and functional test frameworks—is fed into an AI engine, which provides real-time decision support on release readiness. Key benefits of AI-enabled continuous testing include enhanced testing speed and scope, a shift from reactive to proactive testing, and the cultivation of a culture in which all team members are empowered to own quality control.

4. Integration of Continuous Testing in CI/CD Pipelines

Continuous testing has become a vital component of modern DevOps processes, applying automation principles to the entire software testing lifecycle. By automatically executing test cases as part of the CI/CD pipeline, teams receive immediate feedback on the business risks associated with a software release. Integrating automated testing into the delivery pipeline drastically reduces manual testing requirements and the associated risks and costs [9,24-26]. Implementing continuous testing is an important business enabler that supports faster releases without compromising quality or business continuity.

Continuous Testing in DevOps Environments

A fundamental objective of DevOps processes is to enable innovation while minimizing the risks and negative consequences of change. Continuous Testing addresses this goal for DevOps by automating testing at every stage of the DevOps lifecycle, providing immediate feedback on release risks. The key capabilities required for such an implementation are discussed here and illustrated in a typical test automation infrastructure supporting continuous testing. Machine learning with artificial intelligence techniques can play an important role by optimizing the testing strategy in an agile and changing environment where test case execution times and the business impact of various changes differ over time. Testing teams struggle to provide timely test feedback for each change in the software development lifecycle because over time the number of test cases grows exponentially.

4.1. Strategies for Integration

The practice of continuous testing implies testing software with every modification so that errors and defects can be identified early. If a single fault or error is left undetected, it can influence other software components as well as the entire software. This also

reduces the testing time and cost of the product and maintains the quality. Testing parallelization distributes the workload among multiple CPUs, which also saves time.

Continuous testing is attained by performing different levels, types, and types of testing parallel to the development process. The most common test categories used are unit testing, integration testing, sanity testing, acceptance testing, and load testing. Every test is performed by a different role, with unit testing performed by developers and stress and acceptance testing performed by testers.

4.2. Tools for Continuous Testing in CI/CD

CI/CD pipelines are implemented through various DevOps tools, embedding multiple phases of software testing from scratch [27-29]. For each phase of the CI/CD pipeline, there are dedicated tools of testing. These generate multiple reports based on tests triggered by each commit and support the team members in addressing the challenges of continuous testing. Select automating tools, the language in which the code was committed, and type of testing that needs to be performed are used to trigger the tests. Once the code is committed, the commit triggers the test based on team selection. Nightly mode triggers the complete set of tests and creates a report to be addressed by the team. Business stakeholders also support the automation teams with the additional requirement of executing the test on every business release.

During the past decade, many solutions and strategies have supported development teams to enhance software-quality objectives that require evaluation at an appropriate point in software development. As a result, several types of testing have emerged. These include unit testing, integration testing, system testing, acceptance testing, regression testing, performance testing, load testing, security testing, GUI level testing, accessibility, and on-device testing. The DevOps model assists in generating software of higher quality and shorter development cycle with the automated execution of tests throughout the process at different phases of the SDLC. The underlying philosophy of DevOps advocates, "if anything can be automated, then automate it."

5. Real-time Decision Making in Continuous Testing

Real-time decision making in continuous testing must address the numerous challenges posed by automation, integration, and large test suites [30-32]. Considering the size of the test suite and the optimized resource allocation are important factors in achieving client satisfaction for efficient testing. Efforts toward this goal attempt to optimize the selection of the timeframe used for new builds. In continuous testing, delay of streak build completion is a serious concern because sooner the bugs are identified, sooner they

can be fixed. For three consecutive builds, completion of testing duration is minimized to adhere to client demand. Moreover, undelayed time of starting build is equally important. The total time of testing for a release may be reduced by optimizing the start time of the consecutive streak build. In multitasking, a resource cannot be allocated to another task until its original assigned task is completed. Testing resources are allocated to the series of builds accordingly.

Testing resource allocation in a multitasking environment is an equally crucial task to be addressed for minimizing build completion time and testing time, especially when testing of different task groups can be performed in parallel. The decision-making system allocates appropriate testing resources to the streaks in the multitasking environment. During execution, built-in intelligence adapts instantly if there are sudden changes in testing start time and selects the optimized testing resource. Total delay is reduced by selecting optimized testing start time. Finally, all the formulated decision preferences are comprehensively evaluated using the decision-making model.

5.1. Importance of Real-time Feedback

Continuous testing plays a fundamental role in the application of the DevOps process, providing immediate feedback on failures [9,33-35]. Moreover, continuous testing helps release successful builds and identify new features to be developed. Such a process provides rapid feedback regarding application's health. Subsequently, integrating Artificial Intelligence can improve these DevOps continuous testing processes. Applying AI techniques to New Feature Identification is one area where improvements can be attained in the DevOps pipeline.

Rapid feedback is a crucial aspect of continuous testing. Time is a valuable resource, and developers should not have to wait an extended period to learn whether their code changes have adversely impacted the build. Prolonged run times force developers to either work on additional changes before inspection or halt work until test results are available. Building an efficient continuous testing process that quickly indicates the health of the build is of paramount importance yet has not received significant attention in the literature.

5.2. Techniques for Real-time Decision Making

The real-time decision-making techniques implemented in the system rely on a closed loop feedback mechanism. During scheduling, failures of container start, repeated failure of jobs, low resource availability on hosts, and other factors are monitored and fed back to the scheduler before it assigns resources for subsequent runs. Feedback acting on

specific metrics determines the selection of a particular cluster for the next run and the number of containers provisioned in the cluster. With each provisioning request, resource availability on the selected cluster is re-evaluated. Intra-job activities like test execution, job success, and job failure are similarly fed back to the scheduler. The system takes this feedback and schedules the next run accordingly.

Input metrics that define the real-time decision-making process are weighted and assigned to different categories before scheduling a job on a cluster or adjusting the number of containers during execution. These weights are derived from historical and current environment factors and business-critical requirements. Procedures have been developed to ascertain the weight for each factor influencing the decision-making process. Once weights are assigned, category suitability is calculated using a weighted-sum approach and analyzed to ascertain a suitability score between 0 and 1. Threshold values then identify the most appropriate category for the current run conditions. Equations for suitability score calculations normalize individual weights to ensure comparability.

6. AI-driven Feedback Loops

The aim of continuous test automation in a CI/CD DevOps environment is to create a feedback loop that enables software delivery teams to adjust the development and delivery process in response to feedback. The traditional subject of testing has been the product, focusing testing just before deployment to detect defects and ensure quality. But CI/CD enables—and demands—continuous testing of the process in order to deliver the right software, at the right time, to achieve the right business outcome.

Claus Reinke (Enlyft, Chicago) comments: “In traditional QA tests, tests are being written and continually re-used as regression tests. This is a great starting point for automating tests and defining the quality gates for any SDLC. In addition to that, thanks to the availability of huge amounts of data, AI can help with completing the overall testing strategy by providing the possibility of generating highly adaptive test scripts that evolve and train the systems in the normal course of operations. Such an approach helps to mitigate the risk of unknown risk areas and thus unknown test-scenarios. It’s still important, though, to understand what makes sense for each individual use case and the balance of (artificially intelligent) automation and manual control. Not all tests need to be or should be handled by automation.”

6.1. Concept of Feedback Loops in DevOps

A feedback loop is a recognized DevOps practice that provides early delivery mechanisms to identify system and product flaws. The DevOps feedback loop is a cyclical process that addresses problems, applies fixes, and assesses potential risks through monitoring. It involves several feedback channels operating at various levels and phases of the application lifecycle, ranging from coding and integration to testing, delivery, and operations. These channels enable evaluation of the application and its components across the development lifecycle and production environment. Feedback from these channels is crucial not only for pipeline automation but also for business success.

Continuous Testing is among the DevOps phases that receive input from the development team. The implementation of continuous testing varies across organizations based on adopted strategies. Developers activate the continuous-testing phase by including the source application data as input while setting up the DevOps pipeline. Traditionally, continuous testing is a fully automated solution managing all test cases in an automated fashion. Nevertheless, the nature of some applications and the domain expertise of developers suggest executing selected test cases manually. Textual prompts, such as "run all automated test cases for project X" or "run all manual test cases for project X," can also initiate and operate testing activities. The generated feedback is fed back into the DevOps pipeline via a human–machine interface.

6.2. Implementing AI for Enhanced Feedback

One approach to AI integration within DevOps focuses on the Continuous Testing phase. Here, the immediate Test creation and execution, together with the feedback from Test Environment Operations and subsequent stages — the entire left branch of Figure 6.2 — rely heavily on operational data. This Window of Insight offers invaluable information to enable employing feedback loops for Test selection.

To implement these feedback loops, historical and current operational data from the Test Execution and Validation branch are accumulated, stored, and permanently updated within a Data Lake. The accumulated data is gradually made ready for Machine Learning and other data-analysis techniques, facilitating the identification of the most suitable Test-bed configuration for a specific product and the selection and prioritization of Test cases.

7. Challenges in Continuous Testing Integration

Continuous testing integration in DevOps pipelines is crucial for validating each change to the source code base, covering the whole lifecycle from assessing builds to the final release. The integration of artificial intelligence within testing provides the fundamental capabilities needed to implement this, optimizing tools, techniques, and methods applied to testing. A quiet revolution is underway in software testing, with artificial intelligence (AI) and test automation merging to offer immense potential to software QA teams.

Continuous testing integrates early and frequent testing to deliver software faster with higher quality. Techniques such as API-based testing and advanced automation testing frameworks keep pace with agile development, testing every build and, ultimately, every line of code, assessing application performance from all perspectives. In the DevOps pipeline, continuous testing validates code changes from build to release. Deploying artificial intelligence for test automation enables smarter testware that learns from past executions, automatically analyzes results, and reports bugs with minimal human intervention.

7.1. Technical Challenges

Continuous testing in a DevOps environment aims to deliver complete test automation with a streamlined release process. However, several challenges must be overcome to achieve this goal. Testing is a highly repetitive and verbose process, and test automation scripts must be constantly updated to keep pace with rapid code releases. Test code is often less structured, and the emphasis on frequent releases can result in increased bugs due to insufficient regression testing. Given that testing is a cost-intensive, time-consuming activity, complex, rapid, and reliable automation testing often remains out of reach.

Moreover, test scripts are typically designed using simple 'if-else' control statements and are not built for reusability, reducing the likelihood of creating a truly scalable and reusable automation framework. While testing is crucial, it generally adds minimal value to the product and fails to show proportional improvement or growth. The supporting tools for test execution, defect tracking, and agile development lack artificial intelligence capabilities that could enhance test efficiency and provide deep insights.

7.2. Cultural and Organizational Challenges

Ensuring effective testing remains a crucial aspect of every DevOps environment. In a continuous delivery scenario, properly testing developed code significantly reduces the risk of errors or unmet quality requirements appearing in the deployment environment.

While countless technologies and techniques have emerged in recent years to automate testing and enable artificial intelligence within the process, the main problem often lies with company culture and organizational structure. Managers may perceive testing as a boring or tedious task that hampers employee advancement, while team leaders might assign it to recently hired, less trained, or unmotivated developers. Effective testing—with minimal management effort and incorporating AI capabilities—is an ongoing challenge that companies must address for the benefit of their own software testing.

DevOps and Continuous Delivery have produced a dramatic acceleration in release times, making failures an inconsequential, albeit unpleasant, part of the landscape. Restoring proper service is undoubtedly the priority; however, Customer Experience teams also need to analyze the causes and effects of specific failures. Continuous customer failure reports must be addressed—regardless of their origin or impact—to prevent losing valuable customers and damaging the company's business. Although log analysis tools can assist in determining failure causes, they do not empower business teams directly. Allowing testers and Customer Experience team members to introduce their domain knowledge into failure analysis is essential, but current tools and approaches do not support the human perspective on logs as effectively as they do for developers or operators.

8. Case Studies of Successful Implementations

Successful efforts in Continuous Testing in DevOps illustrate the efficacy of current tools and practices, showcasing clear benefits for testing teams, operations teams, and businesses in general. Companies worldwide demonstrate how continuous testing, complements Artificial Intelligence, and Exploit Measures of Prioritization Integrating Relationships enrich practices and extend the usefulness of testing. In general, it helps testing teams control the quality level of software more effectively, operations teams anticipate operational execution security levels, and businesses assess potential financial impacts resulting from poorly controlled operations.

Use of AI on testing teams leads to greater efficiency and effectiveness. AI-based test prioritization methods exploit dependencies and similarities of components and interfaces. They can be based on prior automation of planning, installation, execution, and evaluation to obtain feedback and prioritize test cases over the defined pipeline. For top companies dependent on testing for immediate deployments, the speed of execution can take precedence over the effectiveness of the execution; in such cases, risk-based AI prioritization of testing plays a critical role.

8.1. Case Study 1: Company A

The case study covers Continuous Testing at Company A. It is a large CIS environment for Cloud with many applications. Customer experience approach.

The Continuous Testing part of DevOps has been additionally enriched with AI capabilities to better support the Cloud environment and applications on top of it.

The behavior of 130 business-critical users is analyzed continuously for the latest six months of transactional production data. It is based on the AI properties of self-learning and self-adapting. The AI engine creates and continuously maintains a repository of the emulation models of the users' behavior. Manual maintenance of the repository has not been required for the last four months.

Several hundreds of test cases are executed in the actual CIPD phases based on the repository of the users' model. The AI engine automatically adapts the check points and execures a series of next logical transaction steps. The support of numerous integrated applications makes it possible to cover the complete journey of a customer.

8.2. Case Study 2: Company B

Company B, based in Belgium, supports the DevOps business model by offering a suite of tools designed to facilitate the development process. By supporting the adoption of a DevOps process, Company B enhances software development projects with features such as release management, test case management, application lifecycle management, and continuous integration and deployment.

The integration of AI further enriches these capabilities, introducing advanced analytics, intelligent automation, and predictive insights. Together, these features enable the reduction of manual effort and the enhancement of insights into the testing process—fostering a truly modern development approach centered around continuous testing.

9. Future Trends in Continuous Testing and AI

Continuous testing in DevOps has evolved from automation suites running testers' test cases to self-adaptive, self-trained large AI and deep-learning systems. These systems use real-time functional production data to generate, test, and correct test assets and jobs at one thousandth of the historical cycle times. Continuous testing is a mandatory part of the IT software development ecosystem, as test automation becomes the biggest time expenditure.

The evolution of continuous testing is defined as the repurposing of test assets (test cases and test data) at all stages of the DevOps pipeline. Self-maintenance of test assets is achieved through self-adaptive, multi-layered, deep-learning AI-driven testing systems that examine production logs, jenkins jobs, service and code metadata, and more. The main objective is to maintain test asset hygiene continuously, keeping all test assets valid, updated, relevant, correct, and ubiquitous for the CI/CD test pipeline. The process is optimized across not only the fastest execution time but also business priority, business risk, software risk, and production code correlation. With self-maintenance, execution speed is increased proportionally to the speed of the delivery pipeline. Advances in PiggyBack Testing™ enable Pipelines as Code to be designed and managed continuously, further accelerating DevOps.

9.1. Emerging Technologies

It is necessary to capture the testing advancement trends associated with emerging technologies. By doing so, there is a precise focus on the mainstream related technologies which are shaping the future of testing solutions. Testing applied to the business processes nowadays supports both organisational-orientation as well as built-in orientation. This means it is necessary to consider the testing of business processes that are supported by business components offered by vendors. Testing applied in the next phase for business processes considers their implementation based on the orchestration of business components and business services. Both these trends become empowering from a virtualisation perspective and benefit also from cloud computing.

Defence Testing for Continuous Delivery. Vic Mahadevan, Vice President of Engineering and Development at Accenture, identifies three emerging trends in Cloud DevOps Testing: more monitoring of the deployed application to check what is really working in production continuous monitoring with special metrics related to transactional services. Games and visualisation tools to understand the health, performance, and behaviour of the deployed application. The key testing capabilities to support continuous delivery are: near real-time continuous status of builds without kicking off individual jobs through easy visualisation. developed to satisfy expectations and also for those that have to face the scrutiny of Legal, Safety, and Security aspects.

Emerging Testing Technologies. Testing applied to business process based business component services considers the testing of business services and business services composition as a business process over the virtualised environment created through cloud computing. These trends emphasise that the transition to a hyper-automated future for organisations through robotic process automation, machine learning, artificial intelligence, virtualisation and cloud computing will be reflected by smarter testing enabled by some of these technologies.

9.2. Predicted Developments in AI

Considering contemporary trends, it is likely that artificial intelligence will reshape the approach to software testing. AI can effect substantial modifications when applied to continuous testing processes [36-38]. Continuous testing involves executing automated tests throughout the development lifecycle and integrating the results into the release pipeline. AI contributes by selecting appropriate tests, generating data, diagnosing faults, and automating reports.

The deployment of AI within a DevOps environment has escalated the demands placed on quality control. DevOps facilitates cooperation between development and operations teams, thereby shortening release cycles and accelerating updates [3,39-41]. Continuous testing evaluates applications continuously prior to release and subsequently monitors performance. The on-going collection of reports and associated data thus culminates in a substantial volume of information, posing significant implications for all organizations engaged in software development.

10. Best Practices for Continuous Testing in DevOps

Continuous testing is an integral phase in DevOps and ensures software quality. A major aspect of achieving continuous testing involves leveraging AI techniques, which can help avoid regression and capture exceptions earlier in the development lifecycle. The primary objective of continuous testing is to reduce the Soak Time — the time spent on testing — ensuring that error-related costs remain low despite frequent updates and frequent releases.

The best practices of continuous testing in DevOps include: having a plan for continuous testing; collaborating between developers and operations engineers; establishing a test automation pyramid; committing all code and environment configuration files into Git repositories; making all builds for every code commit; provisioning automated test environments; running automated tests on every build; making aggregated test execution reports; allowing business users to test every release to validate the system; using appropriate performance testing tools to run performance tests regularly; enacting policy-driven approval mechanisms for production releases, and leveraging data-driven and AI techniques to analyze the test results.

10.1. Establishing a Testing Culture

In the last decade, an increasing number of organisations have adopted DevOps for software development. New problems have emerged stemming from an increasing number of deployable changes, continuous deployments and a fast testing cycle.

The term ". . . the testing culture . . . " is an aspect of the software testing process. testers, requirements analysts, developers, product owners, user experience designers, and operation staff test their own work products using various checks, monitors and evaluations in various testing stages throughout the DevOps development cycle. The objective is that all teams are aware that a bug in the production system will cause rework for their own team. They also know that automation facilitates frequent checks and reduces manual work. The DevOps development cycle in the testing culture is both continuous development and continuous testing of production code.

The testing culture awareness provides the team with the motivation for frequent checks and automated tests. It is determined to what extent the teams within an organisation are aware that development generates errors and the higher the position in the organisation the stronger the belief. Furthermore, the testing culture awareness believes that extensive rework is required to correct errors found in the production code and that automation reduces the manual testing effort. The team is conscious of the fact that checks and automated testing can be bothersome in the short term but will ultimately have clear benefits.

In the DevOps development cycle, the phases run sequentially when performed on a particular code commit. The automation of these phases allows them to be performed in the same or nearly the same time it would take to perform a single phase manually. New code commits are integrated and tested regularly and frequently throughout the day.

10.2. Continuous Improvement Strategies

Continuous testing (CT) is designed to detect whether DevOps has been implemented correctly by checking if coding and operations pose risks. Conducting an automated risk analysis evaluates the risk levels of new code commits, thereby prioritizing test cases with low overlap and high coverage. For code changes that incur costs or require additional testing cycles, risk analysis assesses the marginal benefits derived. Risk-oriented CT optimizes resource allocation, mitigates security threats presented by bots, and simplifies the analysis of integrated, combined, and enhanced DevSecOps projects. Incorporating an AI agent to optimize test-case selection and execution further advances the process. CT can be applied to three key areas within DevOps: risk-oriented, cost-oriented, and risk–cost-oriented.

In the risk-oriented approach, an AI agent detects and remedies code bugs during the coding stage, communicating results with developers to ensure operational and testing phases also benefit from AI testing. Risk analysis pinpoints risky areas within code commits, allowing the AI agent to optimize the selection and execution of test cases accordingly. The cost-oriented approach segregates test cases into automated and non-

automated categories; the AI agent then automates additional non-automated test cases, maximizing automation coverage to alleviate reliance on human efforts. Finally, integrating both risk and cost considerations through risk–cost-oriented CT leverages combined risk analyses and test-case automation optimizations. Through this strategy, the AI agent is tasked with selecting non-automated test cases that mitigate the risks associated with new code commits, prioritizing them based on diminished risk levels relative to remaining automation resources.

11. Conclusion

In the world of DevOps, quality assurance is crucial in every stage of the continuous development pipeline, from source code hosting to production deployment and support. Continuous testing is essential for assisting teams in producing software that meets business requirements. Artificial intelligence facilitates the application of continuous testing in DevOps. The integration of AI recruits elevates the speed and effectiveness of quality assurance throughout the pipeline. Intelligent analysis provides a comprehensive mapping of root causes that can impact the outcome of continuous development pipelines. In summary, leveraging artificial intelligence in continuous testing can enhance DevOps practices by addressing quality assurance comprehensively across the development pipeline.

References

[1] Pando B, Dávila A. Software testing in the DevOps context: A systematic mapping study. Programming and Computer Software. 2022 Dec;48(8):658-84.

[2] Pal K, Karakostas B. Software testing under agile, scrum, and devops. InAgile Scrum Implementation and Its Long-Term Impact on Organizations 2021 (pp. 114-131). IGI Global Scientific Publishing.

[3] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.

[4] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.

[5] Cruzes DS, Melsnes K, Marczak S. Testing in a DevOps era: perceptions of testers in Norwegian organisations. InInternational Conference on Computational Science and Its Applications 2019 Jun 29 (pp. 442-455). Cham: Springer International Publishing.

[6] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.

[7] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.
[8] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).
[9] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.
[10] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.
[11] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.
[12] Angara J, Gutta S, Prasad S. DevOps with continuous testing architecture and its metrics model. InRecent Findings in Intelligent Computing Techniques: Proceedings of the 5th ICACNI 2017, Volume 3 2018 Nov 4 (pp. 271-281). Singapore: Springer Singapore.
[13] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.
[14] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.
[15] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.
[16] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.
[17] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.
[18] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.
[19] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.
[20] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.
[21] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.
[22] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.

[23] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.
[24] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.
[25] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.
[26] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.
[27] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.
[28] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[29] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.
[30] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[31] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.
[32] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.
[33] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.
[34] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.
[35] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.
[36] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.
[37] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.
[38] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.
[39] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.

[40] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.
[41] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 9: Artificial Intelligence Techniques for Performance and Load Testing in Modern Software Systems

Partha Mohapatra

AT&T Corporation

1. Introduction to Performance and Load Testing

Performance refers to the ability of a computer system or program to accomplish its designated function within appropriate time constraints for its intended environment. Performance testing of computer system components is the process of verifying that they meet their required performance specifications, which may or may not be detailed in formal requirement documents [1-2]. The primary focus of such tests is to determine whether the time requirements of a system component behave as expected under varying operational scenarios [3-5]. If a performance specification is violated, the system is said to suffer a performance bottleneck. Load is the quantity of users requesting resources at a given time or over a specified period. It is expressed either as the number of user threads that concurrently exercise the system or as the number of transactions per unit interval tested. Load testing is the process of verifying the performance of system components under expected load requirements [6-8]. The magnitude of the load can be either static, when a fixed number of users concurrently exercise the system, or dynamic, when the load varies over the duration of the test. Load-testing activities are vital for detecting performance bottlenecks that surface under expected application usage patterns but not at lower loads. The realization that system load may vary in a time-dependent manner under real operating conditions has prompted the development of AI-based techniques that simulate dynamic load patterns. These methods enhance bottleneck detection while optimizing the utilization of load-generation resources.

2. Understanding Load Patterns

Load testing using AI allows us to simulate a realistic usage pattern with respect to the load of an application. It takes the real-time customer behavioral patterns into account and generates a realistic load on the system.

Load testing with AI can be automated. Normal load testing tools require the testers to figure out the most important scenarios that users typically use on the application. Testers typically select the most important user journeys that represent most of the user journeys [7,9-10]. Load scripts are created for each of these journeys, and the appropriate number of loads VUsers are assigned to each to generate the overall load on the application. Automatic generation of load scripts for the selected scenarios is not supported by the typical load testing tools, so testers manually create the scripts. The journeys and the number of users for each journey are typically selected using customer behavioral patterns extracted from Google Analytics. Testing of some early versions of the application, or a preproduction instance of the application, is typically used to validate the VUser count and load volume for the VUser. All of the steps are manually executed by a performance engineer, which typically takes a significant amount of time, needing the expertise and knowledge of the performance testing tools.

With AI, this manual effort of creating load scripts for all the selected scenarios, including the number of VUsers for each scenario, can be significantly reduced [1,11-14]. The following inputs are provided to the AI model: the last 2 or 3 months of Google Analytics data and the URL for a sample version of the application to be tested, either a preproduction or an early version [13,15-17]. Using this data, the first step is to extract the customer journeys and the number of users for each journey at different points of time, so the volume of load can be generated for each scenario. The customer journeys indicate the sequence of pages the customer has visited in the actual application during each session and the number of customers following each journey. Those journeys covering 80% or more of the customer traffic are selected. The number of customers following each journey varies during different points of time within the period provided. Customer journeys and the number of customers for each journey are extracted for different points of time of one day and for different days.

2.1. Static vs Dynamic Load Patterns

Static load tests use predefined workloads that don't change during the test. These have fixed parameters like the number of virtual users or transaction rates. Because the workload remains constant, these tests don’t need to inject new load or release existing load during execution [18-20]. The load consistently targets the same server functions

and business processes. Test runners record performance metrics, marking tests as passed or failed based on whether predefined Service Level Agreements are met.

Dynamic load tests adjust the workload during testing by adding or releasing virtual users, or changing transaction rates on the fly. In these tests, the HTTP requests target must shift as the load changes. Service Level Agreements determine pass/fail status. However, such tests usually rely on operator input for decision-making, which tends to be slow and reactive rather than proactive. Dynamic tests can be divided into phases with different static load levels, based on targets and scenarios.

2.2. Real-World Load Scenarios

The load in Performance and Load Testing should mimic real-world scenarios, i.e., determining how many users are expected on the system. Moreover, it should set up different scenarios to execute once the expected load is reached during the routine. For example, in e-commerce, the sale of a high-demand product might be modeled. In case of an outage, the system should automatically create user messages to understand the real impact. During big advertising campaigns, the advertising team should be able to register their results, such as opened windows, clicks, etc. In essence, the test can have an expected load of 200 connected users; however, when the 200 connection users are connected, the marketing team should be able to generate clicks, and when an error code is sent, the team should be able to generate messages for the analysis.

3. AI for Load Pattern Simulation

Load can be added at specific points in a test plan using samplers. Samplers send actual requests to the server that is being tested, and can be configured to mimic real-world behavior by sending requests that resemble what a user would generate on a site [19,21-22]. Timers can be used to introduce delays between requests, so that a single request for every simulated user is not sent. Controllers can control the order of requests, or group samplers – for example to pause the test between groups of requests, or repeat a group of requests a number of times. Assertions are used to check whether the request returned the expected response. For example, an assertion can check that the response code was 200, or that the response data contained a word or phrase.

The probability of each action in the test plan should be proportional to the number of times it should run during the test, unless the probability is specified as part of the behaviour being tested (e.g. login requests must only constitute a small proportion of the overall load). A small test with a small load might only run actions once, but these same actions should be run many more times in a larger test with a higher load. Historical data

from test plans can also be used to predict the likelihood of certain behaviour being used, using a service such as Amazon Forecast. This service can use multiple time series from different testers to provide a better forecast of load for an upcoming test.

3.1. Overview of AI Techniques

Several types of AI features can potentially benefit performance-testing tools. Two of the most obvious are the use of AI techniques to identify where bottlenecks are the most likely to exist and to determine the right mix for load tests by modeling usage through behavior recognition. Machine learning also can help testing tools examine the large quantities of data generated by performance tests and identify the best outcomes or the root causes of unexpected test behavior.

Overview of AI Techniques Machine-learning (ML) and deep-learning (DL) techniques use classification or regression to allow computers to learn from and make predictions on unstructured or unlabeled data. Generative modeling enables computers to create new data or content with the same characteristics as the training set. Computer-vision techniques can extract meaning from images and video. Clustering groups data points with similar characteristics.

3.2. Data Collection for Simulation

Recently, a new possibility emerged to speed up the whole process in an automated way without dealing with the schematic view of the database and relations, which makes tens or even hundreds of SQL CASES look simpler while keeping the same level of accuracy. By generating the data using an AI agent based on the original data from production, the system can use the same kinds of real data as seen in production while providing it in a variety of quantities in the testing environment [11,23-25]. It is necessary to extract the data from production and prepare the model input before feeding it to the simulation. There are two options to prepare the input. One is to measure the traffic in production as usual and generate the SQL CASEs [26-28]. The second option is to feed the original traffic to the generative AI and let it generate the list of CASEs according to its keenness to the data, which means it will regard the data as it is in production, distribution, and sequence.

3.3. Modeling Load Patterns with AI

To evaluate the consumer-facing store locator web service's performance, tests utilized the Mechanics workshop application located in the site. Mechanics in 2 days order

around parts for customers with mechanic accounts. The store locator web page caters to customer and mechanic requesters alike. Asynchronous AJAX requests rely on two other services, Store service and Account service, which furnish store and account information. By harnessing AI, users can craft more human-engaging load test scenarios, demonstrating a natural, conversational request flow.

The approach begins with a conversation, steering away from Web API endpoints. OpenAPI specifications still shed light on URLs, methods, and query parameters, while user-supplied textual descriptions define the call's essence. Tools like ChatGPT transform user dialogue into a syntactically correct LambdaTest Performance script, guiding subsequent load testing phases. Requests yield a list of nearby stores, from which the AI chooses a name to fetch stock details. With a budget set, the AI determines a purchase quantity, selecting an account and proceeding to buy parts. The workload model represents a concurrent siege of the tested service by prompts extracted from the conversation. Performance scenarios assign weights to individual requests, controlling their frequency and simulating diverse testing requirements. Calm periods between requests, infused with human-like randomness, further refine the simulation.

4. Predictive Performance Bottleneck Detection

Within the corpus of testing techniques, performance and load testing occupies the unique area of overall system behavior analysis under various conditions, including peak or emergency workloads [29-32]. Common practice employs load generation with baseline measurements, progressively increasing the simulated load to identify a runaway behavior indicative of a bottleneck.

The introduction of the transformer AI model BERT provides a statistically predictive variation of the bottleneck detection process. Combined with Spearman's rank correlation coefficient for time-series analysis, the BERT model can be trained to predict the presence of performance bottlenecks without actual load generation. BERT's applicability to the domain of predicting performance bottlenecks in software systems is illustrated by applying the model to a publicly available data set for CPU usage of a common software system.

4.1. Introduction to Predictive Analytics

Predictive analytics uses statistical techniques—machine learning, predictive modeling, and data mining—to analyze current and historical facts in order to make predictions about future events. The main goal is to improve business performance by forecasting trends and customer behaviors in advance.

Predictive analytics can be described as any method that extracts information from data and uses it to predict trends and behavior patterns. Many people confuse it with forecasting or forecasting models, but the modeling process behind the two techniques is slightly different. Business forecasting is more focused on predicting what is likely to happen in the future at an aggregate level—for example, what will be the market size and growth for this product segment next year? Predictive analytics, in contrast, examines trends and relationships in past data in order to assess the likelihood of a particular outcome in the future—for example, which customers are most likely to respond to my next marketing campaign?

4.2. Machine Learning Algorithms for Detection

Machine learning algorithms offer the potential of more up-to-date scenarios and self-learning performance models to recognise gradual changes in factors influencing the system. Performance and load testers would be able to load test websites directly without having to prepare load test cases [31,33-35]. Instead, the testers would submit a high-level goal, or intent, of the load test scenario, such as macking the performance of the service using the coincidence of sixo Saturdays where the external conditions are sunny and temperatures are above average, with the mention that the goal is to test the website's service under high demand. New algorithms could carry out the entire load test automatically.

The most popular machine learning algorithms belong to deep learning, which are usually used for natural language processing, image classification, or generating music. Performance testing, by contrast, must use other methods, owing to its different nature and requirements. The input data must contain queries written in natural language or test scenarios prepared for offer detection, and the expected output is not merely a classification or generative prediction. Instead, the output is a combination of both types. In the case of offer detection, the input is a word, and the output must provide the classification of the word (whether it is an offer or not) and the missing parts of the description to ensure a complete understanding of the detected offer.

4.3. Case Studies in Bottleneck Detection

Bottleneck detection during peak loads is an area of enhancement for performance testing. Server development and load testing remain a priority in the gaming industry [36-37]. The load-testing server needs to be extensible; any mistake in the feature flag might result in a server crash. Testing the development server of a very high load is essential. Before deployment, load testing a server under heavy traffic is crucial to identify potential bottlenecks.

During load testing, an intelligent bot can simulate user behavior, randomly choosing among various options. From the times it gets stuck or receives errors, it can estimate the chances of the server under test not reacting properly to peak loads. While delivering performance testing, it is important to design the Internal Rate of Return, Investment, and Resource Costs for a project. In API testing, SQL commands are tested to check transaction capabilities.

5. Reinforcement Learning for Dynamic Tuning

The development of AI-based performance testing tools that take load testing to the next level is now underway. Such AI control testing engines can execute a test plan that dynamically adjusts the load on the system based on the behavior of the system—even adjusting the number of VUs executing the plan during the test itself. This automatically maintains an appropriate load without requiring input from a human operator.

This can be achieved by Reinforcement Learning, an area of Machine Learning concerned with how a software agent should take action in an environment to maximize a cumulative reward. The load represents the control input that can be exercised to exert pressure on the system (also referred to as the environment), which responds with a reward signal that helps the agent to learn an optimal control policy.

5.1. Basics of Reinforcement Learning

Reinforcement Learning (RL) stems from behaviorist psychology, in which agents learn to maximize expected rewards by interacting with their environment. The agent learns a policy function mapping environment states (from a state space) to actions (from an action space) that results in the highest expected cumulative reward. Feedback from the environment is the reward function, providing a scalar reward for an agent's current state and action. The environment evolves in discrete time steps based on the Markov Decision Process (MDP). Reward signals are often delayed and not necessarily immediate consequences of the respective states and actions. Delayed rewards are considered using a reward discount factor γ ($0 \leq \gamma \leq 1$). The goal of RL is to find an optimal policy π^* that maximizes the expected cumulative reward.

Deep RL—and more recently, Transformers—have been successfully combined with Monte Carlo Tree Search (MCTS) in the game of Go. Typically, neural networks learn state values and policies simultaneously. These functions serve to guide and reduce the search space of the MCTS, enabling it to play stronger and more strategic moves, avoid exploring obviously weaker ones, and also perform a search deeper than a conventional MCTS.

5.2. Applications in Performance Tuning

For application performance tuning, the AI agent performs a mix of behavioral and content analysis. Before test execution begins, the test engineer provides it with an application ja3 fingerprint (a fingerprint of the client that elicits HTTP requests), and the result of a crawl that captures the desired transaction paths for the test. The ai.loadAgent.NextStep() method is then repeatedly invoked, and the ai.loadAgent.PutObservation() method is used to feed temporal evidence from the application responses. This supports the progression of a walk, emulating a gracefully wandering user rather than one focused on completing individual transactions at maximum speed.

The completion of a transaction beyond a given threshold is regarded as negative behavior, prompting the AI agent to reduce the rate with which that transaction is generated. The AI agent is further configured to recognize specific HTTP response status codes as indicative of negative behavior (e.g., 401, 402). When responses with these codes are received, the current transaction is immediately marked as complete, with the time taken to last response recorded; an associated bin with a “Negative” label is allocated to these transactions, which may be nested underneath individual transactions or routes.

5.3. Challenges and Limitations

When a system should work constantly for a long period, a load test can be done for days or weeks, which consumes more cost and time with the traditional approach. Image arrays can be created using Generative Adversarial Networks (GANs) to mimic real-world noise and small environment change effects to derive realistic variation. The issue is when a great amount of GPU power is needed, and a large dataset is required to train the model.

Stability and reliability areas represent a severe challenge since the presence of random variables and errors reduces the test results' performance. One suggested approach is to generate training data using a tool that can run a load test on the target systems for a certain period, but this approach is limited to systems that are continuously running, and generating enough random variation results can consume a long time.

6. Integrating AI into Existing Testing Frameworks

Considering the use of AI tools within the execution of existing load tests can be an alternative. Such a testing scenario would be similarly enabled by connecting the load testing tool with a virtual assistant geared specifically for load testing, for instance the

aforementioned ChatGPT Bot for k6. In contrast to having the AI generate the entire Load Testing Job from scratch, here the operator would execute and subsequently analyze the test. Commands to launch the test, visualize results, and derive conclusions can then be issued to the AI.

An alternative integration method can leverage the AI capabilities of a desktop assistant. Through a plugin architecture, the assistant can link into the API of the load testing tool or sharescreen with the operator once an interactive session is requested. Again, the integration enables the use of natural language rather than CLI commands or a graphical user interface for all operations involved in testing execution and analysis. Much like in the first option, the AI support expands the hands-on testing experience toward a discussion about the output and its impact rather than focusing solely on the execution as such.

6.1. Framework Compatibility

In performance and load testing, an AI-assisted test-generation tool helps create and maintain a comprehensive suite of test scenarios. Developers, operators, and testers input brief informal requests in their own words, which the tool uses as a starting point. It asks relevant questions to clarify intent and employs code similar to that in the system under test to achieve profound understanding. Ultimately, the tool produces detailed performance- and load-testing scenarios. By clarifying vague or under-specified requests, elaborating terse specifications, and continuously updating the test suite in response to system evolution, the tool maintains alignment with performance and load-testing objectives throughout the development lifecycle.

The tool adapts seamlessly to various scenarios, workloads, and demand patterns, reducing time and effort by automating tedious routine work. It accepts natural language requests from different teams, asking investigative questions that lead to comprehensive input requirements. It supports batch-mode operation, parsing specifications in source-code form and generating scenarios accordingly. Its compatibility with multiple open-source frameworks, such as JMeter, Locust, and Gatling, as well as commercial tools, makes adopting intelligent scenario generation and maintenance straightforward.

6.2. Best Practices for Integration

Because performance testing has similarities to load testing, many tests can be automated by following the same type of strategy outlined under chapter 3, "Best Practices for AI Integration in Load Testing." AI and large language models (LLMs) can assist in generating static checks based on best-practice rules defined within the test case, such

as URL calls, response header values, and response cookie values. However, for more complex queries like SQL injection detection or broken authentication testing, it is advisable to use a static code analysis (SAST) tool or a dynamic application security test (DAST) tool for security testing.

Reports generated from performance tests can be examined using a logging tool or a long-language model, like ChatGPT, for analysis and summary. Despite the fact that model training stops at a specific date and it is actually unaware of any events happening after that date, the model can still be queried for various types of analysis based on the input provided in a test. When examining potential security holes found by scanning a website, it is recommended to search for additional information on the web and verify valid findings with either an expert or the user community.

7. Evaluating AI-Enhanced Testing Outcomes

More and more articles on the impact of generative AI on test automation present the subject from the viewpoint of functional testing, with barely a hint at the beneficial influences on QA's other pillar: Performance and Load Testing. After all, the objective of Load Testing is to establish the performance margins of an application, detecting memory and CPU spikes or leaks, not functional bugs. Consequently, the guidelines for creating test dataset generators with AI--which are the subject of this analysis--allow an additional surprising conclusion: how to create Generative AI prompts to train Artificial Intelligence for Load Testing.

The evaluation of the result is simple: the load times and CPU usage must behave adequately. This performance dimension, although sometimes overlooked, is of paramount importance in Distributed Artificial Intelligence, particularly Dialogue Systems. Natural Language Processing requires an exponential amount of computing and telecommunications resources. As a result, Cloud Providers' efforts toward Green AI are equally well directed to Performance and Load Testing, in addition to the Functional variant. The formulation of Performance and Load Testing Dataset Generator requests with Generative Artificial Intelligence becomes necessary.

7.1. Metrics for Performance Evaluation

One of the challenges of evaluating synthetic Q&A agents involves the selection of appropriate performance metrics. The skill of a human technician is often judged by performance indicators such as Average Handling Time (AHT), First Call Resolution (FCR), Customer Satisfaction (CSAT), or Net Promoter Score (NPS). Qualitative

measures may include user surveys, assessment of communication skills, empathy, politeness, and listening skills.

Measuring the performance of AI assistants is equally complex. Although agents like ChatGPT can engage in discussions on diverse topics and generate responses within seconds, the quality of their answers may vary considerably. For instance, the responses of ChatGPT can be evaluated across performance laboratory tests using the same scales as those applied to human technicians. This provides an alternative method for grading the synthetic agent's answers, enabling examination of answer quality in relation to the average handling time.

7.2. Analyzing Results and Feedback

The results of performance tests are often presented as a set of graphs of throughput, latency, CPU use, and other performance-related measurements versus time. Typical throughput graphs show the number of actions or concurrent users versus time during the test. Latency graphs demonstrate the response time of the system to these actions. Latency graphs may also show the minimum and maximum response times per action. A throughput graph will identify the number of concurrent users for which the system's bandwidth is roughly 80-90% utilized and a latency graph can identify the latency at this point of bandwidth utilization. Together, these graphs show the relationship between latency and throughput. Tables are sometimes inserted to accompany the graphs, indicating times exceeded for particular operations and the number of errors. At a minimum, the following statistics need to be reported: average response time, peak response time, minimum response time, and 90% response time. A good performance test identifies bottlenecks in the performance of live systems by examining the response time between certain operations and the throughput capabilities of the system. Application latency is the factor that drives the transaction cost in the customer's mind.

Load test data can be viewed in a number of ways. Output data can be output in an ASCII format, including pipe-delimited plain-text, comma-delimited CSV, or TAB-delimited for Excel. As one increase the number of Virtual Users, it can cause logon issues as well as high latency that can be seen clearly in a graph. Without logging in and running the performance test scripts, Lucy gave the following recommendation: First, check the number of users/inventory and monitor channels suggested —should be sufficient to carry out the test. In addition, the users per Catalog should be distributed uniformly as listed earlier. Second, create the users and check that the inventory has got allocated to the users. Third, monitor the channels periodically during the creation of users and the channels stability.

8. Future Trends in Performance Testing with AI

AI-supported performance and load testing brings a lot of benefits. Yet, it also has weaknesses and limitations, for example, there is always bias in AI algorithms. Data and dimension are two key factors that increase costs. The future of AI-supported performance and load testing largely depends on the availability of historical data of sufficient quality and quantity.

In addition to the data quantity for training of an AI model, the size of the business and the associated dimension of the application play an important role. Larger companies usually have the necessary data to train a good performing AI model. They also have the resources to run larger dimension tests. Conversely, smaller companies have less data and also less budget. This currently makes AI-supported performance and load testing hardly available for SMEs.

8.1. Emerging Technologies

The massive adoption of digital services and the migration of applications to the cloud have dramatically increased the demand for technologies able to collect and analyze software execution data for troubleshooting, forecasting, and root-cause analysis. In the performance engineering domain, the development of artificial intelligence (AI) algorithms for continuous performance testing, monitoring, and management has gained extensive attention. The resulting promise is to automate time-consuming manual tasks related to performance testing, analysis, and tuning of business-critical applications. Performance testing services can aid in proactively validating application workloads, revealing performance regressions, spotting code-faults expressed as performance defects, forecasting application metrics for capacity planning, and benchmarking application performance. The discussion concludes with an overview of the state-of-the-art in AI applied to performance and load testing and further outlines future challenges for the community.

Continuous performance testing constitutes a set of testing services provided as part of the DevOps process flow of business-critical applications. These services include continuous performance monitoring, analysis, tuning, and forecasting/forecast analysis. The main objectives are to automate the generation of testing scripts, forecasting of application key performance indicators (KPIs) for new unseen workloads and previously nonconsidered resource bounds, and the usage of tests during execution to spot faulty workload/job combinations. Examples of such workloads are unfair CPU usage, excessive usage of memory or storage, or excessive usage of habits sensitive windows such as network or disk.

8.2. Predictions for AI in Testing

Although it is difficult to forecast the future, there are some clues about the use of artificial intelligence in testing. AI already plays a significant role in testing, and in the future, its importance seems likely to grow. Automation will become a daily task for testers who are not specialists in performance or load testing. AI is expected to accelerate UI testing. Testers will employ AI to optimize testing processes, thereby reducing unnecessary testing. Furthermore, AI might be used for exploratory testing by simulating user behavior.

Performance testing is a critical topic for AI systems. While training AI may not be particularly difficult in terms of raw computing power, ensuring the responsiveness of the resulting model remains essential. Testing with natural language is a transformative approach, enabling even beginners to create performance tests. These predictions highlight AI's evolving impact on automation and testing.

9. Ethical Considerations in AI Testing

The utilization of AI systems in testing introduces a host of new ethical considerations and challenges. The potential for AI-enabled testing approaches to increase transparency through real-time monitoring may be offset by a lack of explainability, especially when NLP techniques interpret logs and test scripts. This is an important concern given the highly sensitive nature of some systems that require testing. For example, it may be dangerous to simultaneously perform the tests themselves and monitor the system for performance violations in domains such as an AI supporting chemotherapy decisions. Laws regarding AI, such as the EU's Artificial Intelligence Act, should be carefully considered in this domain. AI testing can introduce a number of new challenges if the AI being tested has the ability to take action in the real world. From an ethical standpoint, it is best practice to perform the tests within a sandbox simulation, where no real world damage can occur as the result of malfunctioning components or new models undergoing testing.

9.1. Data Privacy Issues

Nowadays, all types of testing require data that is as close as possible to production data. That means products implementing data privacy keep hackers from spending a lifetime stealing the database of an enterprise. Securing enterprises also means using data masking algorithms to disguise personally identifiable information (PII) in databases. The data generators will always awake concerns about pollution of the data basis of the results. Nevertheless, encrypted data are the best solution for performance testing as well

as load testing. Performance and load testing are also or even mainly a matter of data security. One can invest billions of GDPR projects and hide all data behind a black wall from the simulation of a real application launch on the internet.

Withcasting mostly hides the realistic amount of data behind pseudo productions, which are relatively different to the original data. Nevertheless, some data are stored in the encrypted files for a longer time and a real production-like use of these files can allow more stable development in all kinds of application testing. Withcasting includes the aspects of hiding a time of down time and an amount of flood of data or storing data too early in the fictitious and withcasted scenario. During the period of war in Europe, withcasting is a great approach to simulate a business-as-usual situation.

9.2. Bias in AI Algorithms

AI algorithms have been found to contain biases, many traces back to the data the system is trained on. News articles citing biases in AI appear almost everyday. Performance testers must understand the dangers of such bias in AI; the goal is to understand how to implement valid tests that account for and analyze the risks of bias. How AI systems may affect the natural diversity of society is difficult to judge. The emergence of biased AI training datasets may affect communities by inadvertently indulging in historical or social prejudices. Preventing biased AI behavior is not trivial during the early days of AI development; however, it is prudent to develop a list of biases in AI for boards and government officials.

The topic of bias in AI has often been found lukewarm or poorly addressed in the area of AI performance testing. However, it is a very relevant concern. For example, in a recruitment agency, if the algorithm has been trained with datasets containing a gender or race bias for a role, it will be reflected in the AI algorithm asthe recruiter directs job vacancies for those roles only to a specific group. Adversarial robustness is another form of AI testing that could be exposed to bias and adversarial attacks. It is imperative that the AI algorithm built for the end solution go through bias testing, security testing, adversarial testing, as well as performance and load testing.

10. Conclusion

AI can take on the drudgery of load-testing at scale while keeping pace with agile development, boosting the effectiveness and efficiency of QA engineers, and keeping gifted testers focused on services rather than a UI. For each challenge that arises in load-testing, it is possible to use AI-powered options today to solve the problem.Now, AI can become a QA engineer’s assistant as well. AI-powered approaches can Helps reduce the

operational overhead of maintaining load-testing scripts. Allows the generation of UI load-testing scripts without requiring the QA engineer to understand the internals of the application. Reduces the need to maintain/QoS monitor load-testing scripts during QA, allowing QA engineers to focus on monitoring the system under test rather than focusing on monitoring or updating scripts.Automation and AI can handle the drudgery of load-testing at scale while concurrently keeping pace with agile development. The solution makes load-testing far more efficient, reducing total test time, and it helps QA engineers be more effective, because they can properly focus on service testing rather than UI testing.

The continuous load testing of newly built software is key to achieving scalability and reliability guarantees in production. Real user traffic is the best surge traffic generator, but it only comes after the software is deployed and the customers of the software can break the system. Load testing enables discovering bottlenecks in software prior to software deployment. The cycle to build a new microservice in Kubernetes has shortened to hours or even minutes. The living QA engineers who ensure the gradual scalability of services under load-testing cannot keep up. The cost of manual testing is extremely high once microservices support complex data flows and engage with several upstream and downstream services. Surging through load-testing using Apache JMeter requires surge traffic generators. Scaling up surge traffic during software testing to the desired extent has become a burden.

References

[1] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.

[2] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.

[3] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.

[4] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.

[5] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.

[6] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.

[7] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.
[8] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.
[9] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.
[10] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.
[11] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.
[12] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.
[13] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.
[14] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.
[15] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.
[16] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.
[17] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.
[18] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.
[19] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.
[20] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).
[21] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.
[22] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.
[23] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.

[24] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[25] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.
[26] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[27] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.
[28] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.
[29] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.
[30] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.
[31] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.
[32] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.
[33] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.
[34] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.
[35] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.
[36] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.
[37] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.

Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-046-9

DeepScience
Open Access Books

Chapter 10: Artificial Intelligence-Powered Software Testing: Challenges, Ethics, and Future Directions

Partha Mohapatra

AT&T Corporation

1. Introduction

Machine learning (ML), a branch of artificial intelligence (AI), enables computers and software applications to "think" and react like humans. Rather than operating according to rigid rules, they "learn" from the data they acquire on their own. AI testing aims to verify that AI models have been well trained and optimized for selection. AI can be applied to many different areas in software testing, such as test case generation, test classification, test optimization, and so on. During testing, AI assistants can help test engineers to create scripts for better coverage of the functionality and do drastic optimizations of the test code and test cases that are irrelevant because of changes in the product. It was reported that around 166 companies worldwide have been building AI test solutions. As research on the application of AI to testing is mature, it is beneficial to understand how AI technologies improve software testing.

There are also certain roadblocks to the application of AI in testing [1,2]. For example, AI algorithms may take a lot of time and effort to build and label the data needed to train the AI models. AI-based systems may also operate as "black boxes" and are difficult for testers to explain. When they are applied to areas such as healthcare or the NASA PATH FINDER mission, the assurance of quality, security, and privacy would become the cornerstones of human trust and confidence. In addition, the application of AI to testing needs to be addressed from an ethical perspective. Unlike the usual humans who design the software, AI-based systems are associated with key characteristics such as autonomy, intelligence, replicability, and self-learning ability. To cover testing the various features of an AI-based system, a new test model should be designed by taking these into consideration.

2. Understanding AI in Software Testing

As the field of artificial intelligence matures, software testing increasingly employs AI and machine learning [3-5]. AI methods range from classic heuristic algorithms to deep neural networks (DNNs), used throughout the software testing lifecycle—ranging from fuzzy test oracle decision-making through test case prioritization and generation to test execution [6,7]. Despite the evident potential, progress in developing AI methods for software testing is problematic. To maintain the correctness of the learning process, there is a need for balance during data preprocessing and model training. However, the creation of an ideal dataset is challenging, especially in terms of completeness, objectivity, and genuine representation.

Research trends in the application of AI for software testing reveal a growing number of new methods proposed, along with heightened concerns regarding the limitations and pitfalls of AI. Within the Software Engineering community, governed by The Software Engineering Code of Ethics and Professional Practice, it is paramount to endeavor to "artificially intelligent" software testing models that are competent, ethical, reliable, reproducible, replicable, useful, and ideally also informative. The utility and trustworthiness of developed AI models depend fundamentally on competent problem formalization, appropriate benchmarking practices, and proper acknowledgment of potential limitations and shortcomings. By addressing these aspects, the concerns and risks associated with deploying AI in software testing can be better managed and mitigated.

3. Accuracy in AI Testing

The accuracy of AI testing is an important consideration, which depends on the test data in the case of supervised learning [2,8-10]. Because better real-world data lead to better solutions in such cases, it is important that the data represent the wide variety of scenarios the software is expected to handle. Apart from just the base data, a wide variety of test cases ensure the correct and complete functionality of the software, and ensure the software can handle as many situations as possible. Therefore China reportedly created ten thousand test cases to verify the effectiveness of their AI-enabled, autonomous subways. Testing against the base dataset guarantees accuracy, whereas testing against the test cases guarantees robustness.

Another aspect to be considered is that the behaviors of two different AI programs may not be the same for the same ML task in the real world. Hence, AI-generated test cases

may not work for different AI programs. Although certain test cases are common, the non-functional ones may vary. Thus, once the AI software is developed, it has to be tested once again using existing datasets and test cases, thereby shifting the AI development towards traditional software processes.

3.1. Defining Accuracy in AI Contexts

The notion of accuracy plays a pivotal role in AI testing. For example, language models are often employed in software testing to develop test cases and scripts, evaluate the software, and create other testing components [1,11-12]. In these scenarios, it makes sense to deem models "accurate" if they excel in those approaches. However, one must question whether it is appropriate to label a model as more or less accurate based on the quality of its outputs, as other, equally valid approaches might rate different models differently. Indeed, the concept of accuracy lacks a definitive interpretation.

An alternative perspective defines accuracy as the degree to which the produced result aligns with the input. In language models, this translates into how well a model replicates the provided words and their order within a prompt. In software testing, it could mean the extent to which a generated test case or script corresponds to the received prompt. This interpretation resembles the "ground truth" approach commonly found in various domains.

3.2. Measuring Accuracy in AI Testers

Measuring the accuracy of AI in testing is challenging because it is difficult to characterize all possible bugs within the software under test [13-15]. At the moment, practical attempts at improving testing with AI must also explicitly limit the scope of the problem. OpenAI's recent reports on using GPT-4 as a tester are therefore explicit about measuring performance on a subset of bug types.

The practical need to narrow the focus of an AI-driven tester is clear when considering, for example, the types of bugs that GPT-4 will not find in the current OpenAI API implementation. Certain bug categories are ruled out by the simple exploratory testing approach used and the need for a tester to operate with only a very limited context. Other types are less obvious, but also excluded. These omitted classes serve to highlight, by contrast, the categories of bugs that OpenAI considers, not only for GPT-4 but also for a general-purpose tester. A reasonable approach to measuring the useful accuracy of an AI tester, therefore, is to explicitly restrict the types of bugs being considered to some well-defined subset.

3.3. Challenges to Achieving Accuracy

To improve the dominance of AI/ML testing techniques in software development, achieving accuracy—and harnessing the pattern recognition abilities of ML—remain prime challenges. One major barrier to accuracy is that incorrect output by a ML model is often due to an inadequate ML model architecture rather than an ineffective AI testing strategy [16,17]. Accurately analyzing the appropriateness of an ML architecture represents an entirely different area of future research for AI testing. Attention must also be paid to the fact that training of AI and ML models for software testing is an open-ended research area in its own right; the quality of the training dataset has a significant impact on the effectiveness of AI for software testing [12,18-20]. Therefore, the degree of accuracy achievable will ultimately depend on how the models are trained. Finally, the intelligence of an AI system is constrained by the knowledge it possesses, which is drawn from its training data. AI systems cannot generate new knowledge beyond what is contained in their training set, thereby limiting their ability to handle novel testing scenarios.

4. Transparency in AI Testing

Transparency generally encompasses the data used to train the model to the decisions made by the agent, often evaluated through empirical experiments. Several interpretability methods are available to explain individual model decisions. Many companies regularly commission audits of their AI operations. For AI systems allocated to public authorities and governmental bodies, explainability should not be limited to providing explanations for decisions but should also cover the essential elements of the model's design and functioning. The decision should be able to be explained in an intelligible form. Decisions should also be perceived as fair, which means explaining the outcome of decisions and the rationale behind AI system operations capable of performing from different perspectives.

All aspects previously discussed with proper validation, privacy preservation, and transparency measures should be included in the audit report and evidence from the analysis of generated test cases, test oracles, and testing results. The implementation can be integrated with Trojan software or artificial backdoors that activate only when specific conditions are met. Hence, the generated test cases should be suitable for evaluating the system for TROJANS. Test cases generated cannot exceed the scope of requirements, adhere to testing rules defined by objectives, approaches, industry, and project, and they must be ethical.

4.1. Importance of Transparency

Transparency in the context of AI is a complicated and elusive goal, more akin to modelling explainability than it is to actual transparency [21-23]. Not only does it require an explanation of why and how the AI is making a decision, but this explanation has to be in a form that is useful for the recipients of the explanation. While a technically precise explanation is relatively easy to produce, it is far from the most useful form for many recipients. For instance, an explanation accurate to all the model's internal workings may not be easy to understand, nor immediately suggest numerous areas of improvement. In the case of AI, more transparency often means that the user or beneficiary of the output can also provide feedback, which fundamentally reshapes the relationship between the AI and its users.

Thus, thought must be given to the process of explanation, including who both ought to be and is likely to be the recipient of an explanation [24,25]. Classes of recipient benefit from explanations of different types for varying purposes. While these purposes may differ, the explanation should be as technically accurate as possible without compromising the usefulness of the explanation to the recipient. It is, however, desirable that the accuracy of the explanations is reduced in such a way that the information presented is technically accurate, and not misleading. The desired information therefore varies by what question of necessity the recipient seeks to answer. In the case where the fuel that keeps the AI running is data, these questions can be classified by what packages of data can answer them: Process Questions: How did the AI produce answer x? Effect Questions: What effect has the AI produced on decision-yielding data? Tactical Questions: How can the AI be manipulated or convinced to choose decision-x? Ethic Questions: How ethical is the AI's decision-making?

4.2. Methods to Enhance Transparency

Algorithmic transparency describes methods that make the functioning of an AI system understandable for humans [26-28]. Transparent Approaches describe the detailed outputs of the model—its behavior alone or combined with information about the model's internal workings. In contrast, Explainable Approaches use transparent models to explain the outcomes of an arbitrary model. Explainable Approaches include any model with a decision boundary that a human can follow or understand.

Explanations of model predictions include local surrogate models approximating an arbitrary black box with a transparent model, saliency maps identifying areas of the original input that were crucial to the prediction, and counterfactual explanations describing the minimal input change that leads to a different model prediction. Model-intrinsic explanations include interpretable-by-design approaches that enable

straightforward explanation of predictions, either by design of decision boundaries or by aligning models' decision-making with domains of human expertise, and strategies for explanation leveraging automated formal verification. Model transparency refers to the detailed architecture of the model, e.g., the number and type of layers in a neural network, the dataset used for training it, the training algorithm, and its hyperparameter settings, as well as information about its performance in relevant application domains.

4.3. Case Studies on Transparency

Two earlier studies, one on embedded autonomous agents (A1) and one on AI systems for software testing (A2), illustrate specific challenges of transparency. “Epistemic opacity” centers on the temporary intelligibility of system actions by humans, for example the complexity of causal chains within the system. By contrast, “alien agency” involves attempts to comprehend the ‘mind’ of an agent from the outside, formed without the direct experience of being the agent oneself. Embedded autonomous agents encapsulate all pertinent knowledge about their operating environments based on their own sensorimotor experience, but might act in surprising or ‘alien’ ways. Yet, many significant psychology and behavioural ecology questions address solutions to the transparency problem and part of the rationale in studies of animal intelligence is how similar or dissimilar different species appear from the perspective of an external observer.

A1 compares approaches to transparency from the two disciplines and considers their applicability to systems of embodied AI. Recent findings suggest that establishing behavioural alignment to shed light on alien agency is at least equally important for such systems as intelligibility probes into the epistemic opacity of autonomous systems [29-31]. Philosophical questions about secondary consciousness may not be amenable to an answer, but the reader is encouraged to consider that the agent itself is forced to attribute an inner mental life to others in order to explain their behaviour—alien agency will not be possible for conscious beings.

Transparency in AI-based test automation tools raises a different range of issues. A2 is set in the domain of continuous deployment at scale. Such approaches aim to engineer the testing and QA processes by increasing efficiency through automation, often realized by building novel tools. They typically employ AI techniques to create self-learning tools that provide next-generation capabilities like dynamic prioritization. Transparent systems help build meaningful trust and produce valuable suggestions. This study finds that practitioners appreciate when recommendations or actions generated by a tool are explainable and interpretable with sufficient context—intelligible modalities can range from a merely local, ephemeral, and surface-level understanding of functionalities to a more global, persistent, and deeper comprehension of a tool's inner working.

5. Trust in AI Testers

What makes people trust an AI tester? To answer this question, the human testers' perspective on trusting AI is considered, especially since the bias those AI models have might affect the testers' judgement on the testing outcomes. Trust therefore needs to be addressed specifically in relation to AI bias.

Trust has been widely discussed in relation to AI using various terms such as distrust, reliance, dependence, credibility, expectancy, and adoption. Covaci and Teuteberg show that in the context of decision-makers—such as AI based software testing approval—the terms trust and credibility are the most commonly employed constructs in AI research. In essence, credibility focuses on the characteristics of an AI model that influence an individual's trust in the system, whereas trust represents an attitude of the individual toward an AI model. Maintaining trust requires understanding the credibility of testing outcomes by an AI tester. When an AI tester is seen to be politically acceptable, professionally consistent, and capable of executing the testing task through self-confidence manifestations, it is more likely to gain credibility and consequently the trust of its human counterpart. To establish the ground for trust, the subsequent experiment specifically investigates how an AI tester's characteristics affect the human tester's trust in testing outcomes.

5.1. Building Trust in AI Systems

Trust in AI systems is the belief that they will perform accurately, fairly, safely, and reliably at a level that satisfies users. General trust in advanced systems correlates with users' perceptions of their accuracy, fairness, and safety. Also, people tend to rely more on AI-guided decisions as their trust in these systems grows.

Accurate AI evaluations proposed during early AI development stages tend to be accepted by the majority. However, accurately estimating quality without deeper knowledge of AI development or the task itself remains challenging, which may wrongly steer the efforts of business leaders. Consequently, the transparency of AI systems must not only provide comprehensive explanations of operational modes but also elucidate the procedures underlying the construction and testing of the AI technologies themselves.

5.2. Factors Affecting Trust

Trust is "a psychological state comprising the intention to accept vulnerability based upon positive expectations of the intentions or behaviour of another" and plays an important part in relations between people, institutions and even societies [3,32,33]. Trust is therefore important for both users and providers of testing services and AI testers should clearly explain what they do, how they work, how accurate they are as well as what their limitations are, just like any human tester would do. Information is what governs the development of trust and as such better transparent systems are more likely to be trusted. Artificial Intelligence in software testing comes with several benefits and drawbacks. Being able to fully trust an AI tester is not a trivial task, but past research on trust in AI implies that it is achievable. However, different AI systems behave differently and therefore influence trust in different ways. Autonomous systems therefore need to satisfy various factors that influence trust.

The role of the software tester will not only be to work with the AI tester, whether it is a fully autonomous or a slight enhancement, but also to reveal the weaknesses and strengths of these testers by continuously re-evaluating their performance. Testing is a special domain where these systems are the testers and they are therefore expected to reveal their own errors and limitations. These factors influencing trust nevertheless provide a helpful guideline during the implementation of an AI tester in order to obtain a more trustworthy system. While an AI tester ideally would be able to autonomously test any system, trust will equally be a concern for operators and management when using AI for testing services.

5.3. User Perceptions of Trustworthiness

Research shows that different factors influence human users' perceptions of trustworthiness in AI systems, such as the general attitude towards AI, the AI system's performance, and the AI system's transparency. More specifically, the effect of system transparency on trust was found to depend on the user's general attitude towards AI, in that higher transparency increases trust in AI for users appreciating AI systems (pro-automation) but decreases the trust in AI for users dismissing AI systems (anti-automation).

Besides transparency, the importance of the AI system's performance with regard to trust was also examined, finding a stronger influence for good performance than for poor performance of the AI. Similarly, the influence of transparency on trust has been discovered to depend on the AI system's performance, i.e., transparency matters more for good AI performance.

6. AI Bias in Testing Outcomes

AI’s Opacity, Emergent Properties, and Their Implications for Software Testing

An AI system is opaque when it becomes difficult or impossible for humans to understand how the system arrived at a particular outcome [4,34-37]. Many AI use cases produce an unexpected outcome, making it impossible to understand or reproduce it later. Those properties of opacity and emergence make many AI systems unsuitable for some software-testing tasks, such as test oracle, benchmark, complexity, and security testing.

Test Oracle and Benchmark Testing

Testing ideally requires a reliable and provably correct test oracle, which is typically lacking or expensive to produce. Heuristic test oracles are necessary but significantly less efficient and incurable when faulty. Benchmark testing aims to test the SUT against a list of representative, known input/output pairs that assess the ability of SUT to correctly handle such cases. Test oracles require domain experts close to the SUT, which makes them impractical and expensive. If the SUT handles the input correctly, the answer will appear in the benchmark correct answers—a large bulls-eye. In the absence of a test oracle, AI supports test-case filtering, deletion, generation, and selection. Because the inputs and outputs cannot be verified, neither system correctness nor correctness of outcomes can be ascertained.

Benchmark testing aims to test the SUT against a list of representative, known input/output pairs that assess the ability of SUT to correctly handle such cases. If the

SUT handles the input correctly, the answer will appear in the benchmark correct answers—a large bulls-eye.

Complexity and Security Testing

If the SUT begins to exhibit unexpected emergent properties, the testing implies the existence of a security or complexity concern. The inability to ascertain how the test outcome occurred presents a significant problem.

6.1. Understanding AI Bias

AI models are trained using data; if the data contains biases, it's probable the model will learn them. For example, predicting job performance based on past data can be problematic if the dataset contains demographic bias. AI models can also manifest bias in subtler ways, such as focusing on particular features. An author-attribute classifier trained on dialogue from 19th-century plays, heavily weighted toward gender classification, might associate possession of a sword with masculinity due to the era and genre. If during testing, it encounters a female character wielding a sword, it may erroneously classify her as male. Similarly, a model categorizing an image of a mobile phone and a bowl together might label the image as either 'mobile phone' or 'bowl' but is unlikely to identify it as 'mobile phone with bowl,' highlighting that models tend to focus on one label per image even in multi-label situations.

AI models sometimes make simple errors that seem implausible for humans, like misidentifying objects in well-lit conditions or mistaking a silhouette for a shark. These errors, involving what humans call 'common sense,' suggest that AI lacks a foundational understanding shared by performers of the task it's trained to emulate. Perceptions of model mistakes can be influenced by explanations such as heatmaps showing focus areas, but even these explanations may not always be sensible. Consequently, the explanation generation method should be thoughtfully considered to ensure it is ready for practical use.

6.2. Sources of Bias in Testing

Sources of bias in testing can be traced back to the wider software development life cycle. Testers' own personal bias can be manifested in the testing strategy—in the selection of test oracles, input generation methods, and output evaluation (Adams and Hu, 2021). Inner sources of bias in AI testing arise from the aspects of the input and the model under test, as discussed in the subsequent sections. External sources of bias occur when the output is evaluated using a biased ground truth, often influenced by other datasets used in training an oracle.

Input and training data bias can arise from numerous factors, including unrepresentative samples, changes in class distinctions, inaccuracies in the data, and violations of the independence and identically distributed assumption. In the inner source, a biased set of inputs can lead to a wrong assessment of the model's accuracy or robustness, while an outer source of bias emerges in an evaluation oracle trained for the ground truth or golden descriptions. An assessment of the bias in models often overlooks the impact of bias in the test data; indeed, a test dataset should itself be tested for the presence of bias. Imbalanced or incomplete test data will prevent or stall the process of identifying model bias. Another source of bias who influence software testing within a team is a "groupthink" bias, where the testers' choices are heavily influenced by software developers' perspectives.

6.3. Mitigation Strategies for Bias

Bias mitigation techniques can minimize both direct and indirect bias. Direct (disparate treatment) bias occurs when features containing sensitive and legally prohibited information are used explicitly in predictions. Indirect (disparate impact) bias might arise when features strongly correlated with protected attributes drive outcomes in a discriminatory way. Techniques such as pre-processing can detect and obscure sensitive features prior to model training, in-processing methods can constrain models to satisfy fairness metrics, and post-processing approaches can adjust predictions to achieve parity.

Research within software testing has proposed various methods to reduce bias. For instance, the Aequitas toolkit is integrated into a deep learning testing framework to evaluate bias in facial recognition systems. Other systems automatically generate test inputs to measure discrimination against protected groups and highlight avoidable discrimination. The methodology proposed by de Beer aims to mitigate bias in test oracles arising from potentially discriminatory historical data by automatically identifying and removing features that induce bias during test execution.

7. Ethical Considerations in AI Testing

Artificial intelligence (AI) applications must take into account legal, ethical, and procedural regulations to be accepted by society. The appearance of incidents causing property damage or personal injury has raised the question of legal obligations and responsibilities for AI developers who have implemented the components in the products. Ethically, the use of AI applications must eliminate or at least reduce the inherent bias in data. Thus, AI applications must ensure that the model behaves securely and in accordance with society's moral standards. From a procedural point of view, AI

applications must check that the model will not incite users or the public to behave perversely.

Bias in AI arises during dataset development, dataset preparation, data sample labelling, and the learning stage. Solutions involve balancing datasets with appropriately mutated images containing variations in scale, viewpoint, and background. It also involves efficient labelling by which labelers receive the same instructions for data annotation. During the learning stage, bias is eliminated by aggregating information, using techniques such as learning from positive and unlabelled data, cost-sensitive learning, and adaptation and re-weighting distributions.

7.1. Ethical Frameworks for AI

Introducing AI to the testing phases of the software lifecycle requires the understanding of both AI technologies and testing. Ethics are a key element to consider when designing software systems to improve the quality of products and services and thus, the wellbeing of the society. Some moral principles (e.g., automation for good or feedback and self-correction of automated systems) should be specially considered when designing software systems that utilise AI techniques.

Ethical frameworks are proposed to help the integration of ethical considerations in the design of AI systems. Yet, the application of ethical principles to specific software systems is a complex process. The increase in AI demands ethical considerations in all areas of software engineering, especially when the AI terminals are software test phases. Software testers need to become aware of the societal properties of AI to better control the operation of AI systems.

7.2. Impact of Bias on Ethics

In recent years, studies on bias have grown considerably. Bias in an AI system can emerge from any party or artifact involved in product creation and use. For example, if the annotators who label the training data belong to a group different from the AI product's target users, the labeling can lead to bias in the AI application. Bias can also occur because of the developer's background or the system's designers, the data owners, or even the testers. Moreover, the field of software testing can also be biased, as it depends on the domain and the available techniques. Bias in AI is prone to be detrimental to society as a whole. For instance, applying AI in healthcare or creditworthiness testing without addressing bias can have far-reaching and long-lasting repercussions that violate human rights. Researchers have identified bias and ethical concerns associated with AI-based test automation tools, such as unnecessary job loss, capital punishment, and unfair

treatment of citizens. In recent years, there has been growing interest in the ethics of AI and various approaches for testing AI bias. However, ethics optimization has often been considered a sub-goal of general bias testing research.

7.3. Regulatory Perspectives on AI Ethics

Governments and regulatory agencies are becoming increasingly aware of the ethical challenges raised by AI. The World Economic Forum has launched a project to determine "how best to incorporate ethics into AI products, policies, and institutions through global, multi-stakeholder insights into the social, economic, and cultural implications of AI". A recent report by the European Group on Ethics in Science and New Technologies emphasizes the need to design AI technology to be

compatible with fundamental rights, including equality, non-discrimination, and privacy. It calls on the European Commission to assess legal, ethical, and societal implications of AI and robotics and to create an

"appropriate ethical and legal framework" in Europe. G20 leaders have acknowledged the importance of AI but have underlined the need to respect human rights and diversity. The UK's House of Lords

has argued that regulations governing AI and robotics are required to protect privacy, prevent discrimination, and safeguard employment. The Association for the Advancement of Artificial Intelligence is developing a code of ethics to guide the work of its members.

8. The Future of Autonomous QA

A plausible vision for the future of autonomous QA is emerging. Some consider the idea of moving away from low-level, insanely detailed, repetitive, and borderline boring tasks. They envision an AI-led QA agent capable of deeply understanding the product, generating relevant tests, autonomously executing them, synthesizing results, learning from their successes and failures, and writing complex hub-and-spoke tests for less trustable components.

The hypothesis suggests that AI will be a game changer and will reach very high levels of autonomy. Certain autonomous tasks could be completed at a fraction of the cost of similar jobs performed today by humans, allowing human testers to shift toward more creative and exploratory assignments, as well as jobs requiring human judgment. Both humans and AI could learn from each other, uniting in a complementary final product

that leads to safer, more robust software, robust against uncertain, irrational, and random human operative patterns.

8.1. Trends in Autonomous Testing

The use of AI to develop autonomous software test systems has attracted interest. Automation in software testing has been discussed for decades and platforms using AI and machine learning in testing were developed over the last two decades. Recent interest has been fueled by the need for fast testing cycles due to the rapid release of software to the market (for instance, daily or weekly). Getting rid of manual testing requires decision-making capabilities in the test system, including when to run tests, what tests to run, how to increase test coverage, and when to stop testing. Using AI in software testing brings new challenges related to AI, alongside existing challenges regarding testing that have not yet been resolved.

The literature mentions several benefits gained through the use of AI in software testing, including the capability to cope with complex, data-intensive activities, accommodate changes in requirements, provide faster feedback, achieve higher risk coverage, and reduce time and cost. However, there is little discussion of the potential negative impact or risks of using AI. These types of risks and threats are also generally neglected in the emerging industry of testing companies that employ AI, along with the testing of AI itself and the ethics around it. To move towards more responsible autonomous testing systems, the relevant communities need to work on answering questions such as whether AI in testing is responsible or if autonomous testing tools are reliable enough to manage software with significant effects on people's lives.

8.2. Technological Innovations

Test automation is a prerequisite for continuous software testing and is an integral part of DevOps. However, in many cases, it is found to be difficult to achieve. Komariah explore the relationship between DevOps practices and the automated release process, highlighting challenges and offering guidance for organizations in their DevOps journey. They identify that while automated testing is an important enabler of DevOps, a key factor affecting test automation is end-to-end (E2E) test execution that requires access to all system components used in a business flow.

Cotroneo present an understanding of research challenges and steps necessary for the fusion of AI and software testing. AI-powered testing presents new opportunities by reducing barriers to both adopting continuous testing and expanding it with respect to code coverage, input space exploration, and coverage of use cases through search-based

techniques. This undertaking opens the door to new risks and ethical concerns that require careful handling. The advent of generative AI introduces novel challenges related to data leakage and intellectual property rights. Villi propose a comprehensive framework for assessing the risks and benefits of with-the-tool automation, acknowledging that external AI tools complicate assessment due to factors like data leakage and training data.Figure of the original study illustrates a top-level map of AI in software testing.

8.3. Potential Risks and Benefits

Šimić and Vujošević offer a comprehensive analysis of the risks and benefits of using AI in software testing. The discussion they present is well worth reproducing in full, given the importance, breadth and depth of coverage, and the microeconomics perspective.

“Potential risks may include: (1) undesirable short-term effects: security, safety, and compliance evaporation caused by the wrong use of AI/testing knowledge and the overwhelming new debugging challenges of combined code and Test AI faults; (2) bad long-term effects: misallocation of budgets and testers, overconfidence in AI testing, internal attack vectors, and collective or individual demise of testers and testing academics; (3) AI attitude risks: ending up as a tester’s slave or a tester’s task manager; and (4) fundamental risks for AI and AI Testing: the consequences of choosing the wrong approach or building PHATE. Potential benefits arising from the integration of AI in testing can be categorized according to the tester’s skill and seniority into: (1) the art of middle managementâ€”using ATL (Anomaly Testing Language) in full or in part; (2) the tester as a developer at heartâ€”leveraging AI for distribution, selection, enumeration, prioritization, and engine creation (DSEP); and (3) the skills of a senior testerâ€”being supported by AI Testing in a fourfold manner (AI4).”

9. Human-AI Collaboration in Testing

The rising role of generative AI in test generation and maintenance raises several challenges: an overestimation of the quality of the generated software tests, a lack of expertise towards AI-based tests, or the misuse of generative AI tools in the testing processes. They advocate human-in-the-loop support for generating test inputs by combining the strengths of engineers and AI to assess the criticality of their applications. Yet, their work considers the test criticality assessment as done by just one human test engineer who works together with the AI.

Humans generally do not work alone; they organize their capabilities and plan their activities according to the resources available. Moreover, the software testing organization is intrinsically unpredictable because it depends on the highly dynamic behavior of tens of human practitioners, including their interactions and specific knowledge. Indeed, software testing should be viewed as a process governed by a human-in-the-loop feedback orchestration toward test excellence. However, to date, no approach is able to evoke the feelings and 6Es characteristics of human testers. Nevertheless, feeling elicitation, especially the elicitation of testers' software testing experience through their feeling, is vital for testing quality assessment and gap identification.

9.1. The Role of Human Testers

The complex process of software testing at an enterprise level combines machine intelligence and human intelligence, adopting the characteristics of the "Human in the machine" and "Machine in the human" paradox. The "Human in the machine" paradox refers to the fact that the underlying mechanism powering AI software-testing solution is the human brain. Until the advent of strong AI, the human brain remains the only known system capable of supporting the complex operation mode necessary for projects as complex as using AI to test AI in the software-testing context. The essence of the "machine in the human" paradox is the two-way symbiotic relationship between human and machine, where the machine helps humans become enhanced or augmented in processing information. Humans are constantly augmenting themselves through a feedback loop fed by AI products.

Therefore, the human-in-the-loop AI testing method is necessary for the software-testing context, especially when the tested systems are sensitive, time-sensitive, and life-threatening in their usage. For instance, medical robotics require the human-in-the-loop AI testing methodology, as the fidelity of the final conclusion from an AI-based testing activity may change the life expectancy of the human user of the AI-empowered robotic assistance in medical procedures and their consequent outcomes. Such semantics also apply to ensuring safety and security for the users of a self-driving car. Human insights and prior experience become invaluable at any stage of AI-powered software-testing workflows. Specifically, the conflict between time commitment and analysis depth on either side creates an urgent need to differentiate time-sensitive and non-critical testing areas from those that are less time-sensitive but risk-prone. Maintaining this balance requires experience and specific application domain knowledge.

9.2. Integrating AI into Testing Teams

Testing should function as a team, with human testers and tools assigned their own strengths. Machine learning can be a serendipitous by-product of an existing process, with the generated model used for guiding exploration. For example, a bug prediction model generated from product metrics can hint towards which areas of the product may be particularly sensitive or prone to defects, guiding the testing effort. The model is generated from product metrics (code dependencies), not from past testing or coverage data. It is then used for guiding exploratory testing, rather than as a measure of product quality.

The breach of contract dissatisfaction expresses a social preference for AI outcomes that agrees with human preference. In addition, the assigned bargaining power of individual stakeholders can also be reflected in the reward. For instance, the content provider in the above example expects reimbursement, which can be implemented by highly weighting the penalty of unrecognized copyright content. When applied to different stakeholders of the same Smart TV system, AI management is likely to generate different optimal policies, resulting in a social dissatisfaction bias. If several problems come together, the resulting AI may well be classified as a psychopath.

9.3. Best Practices for Collaboration

In many organizations where AI-based testing is being tried, neither the business nor the AI teams are experts in testing. Their focus lies instead on deployment and automation-as-a-service, respectively. Once an AI-based test event is triggered, the remaining stages of bug reporting, tracking, fixing, retesting, and closing involve largely human-intensive work. With test maintenance shifting to DevOps, the focus of test automation tools has also moved from providing action-oriented recommendations to complete bug-resolution solutions.

This new focus leads to a rethink of traditional test maintenance tools into artificial intelligence systems for business support. Companies like ChatGPT and Ada are building AI systems that engage in human-like conversations, assisting users in tasks such as bug resolution. Without such collaborative chatbots, the synergy necessary for continuous integration, testing, deployment, and maintenance remains incomplete. In the absence of a collaborative test-maintenance framework, the promise of test automation remains merely an automation-as-a-service solution.

10. Future Directions for AI in Software Testing

Recent years have witnessed a massive growth in applying artificial intelligence (AI) techniques to software testing. As AI-related technologies and concepts have matured, it has become clear that they create both opportunities and challenges in three major areas: software testing, software engineering, and AI systems. This section provides an overview of the current state of AI-assisted testing and testing of AI systems, considering the phases of the software development life cycle in reverse order, and outlines directions for future work.

The increasing use of software agents, service-oriented computing, and the Internet of Things has made it impossible for testers to comprehend the whole system's structure. AI can play an important role in the testing of these software systems—guiding, controlling, or even performing certain tasks in the testing processes. At the same time, current software engineering paradigms alone do not provide effective solutions for AI systems, making the testing of AI systems the next frontier. Moving from traditional structured programming toward an AI paradigm requires reconsidering the way the software testing community tests these future intelligent systems. These observations indicate multiple future directions in AI for software testing.

10.1. Emerging Technologies

The advent of new technologies such as Artificial Intelligence (AI), Machine Learning (ML), and Big Data has led to the development of numerous applications that utilize these technologies. While the development of AI-powered applications often aims to save time and effort and increase efficiency, these technologies also significantly influence Software Testing. AI can be effectively implemented in Software Testing to manage huge volumes of historical data generated during the software development lifecycle.

An enormous amount of testing data is produced throughout the development lifecycle of a software product, beginning with requirements analysis. This data encompasses relations among various testing elements and is stored in unstructured, semi-structured, and structured formats. Testing element categories include activities like test planning, test design, test execution, and test reporting. AI has the potential to simplify the management and handling of such voluminous data.

10.2. Predictions for AI Development

The most reliable prediction about the future of AI is the impossibility of any reliable prediction. Nevertheless, there are topics that every observer of the field is expected to

comment on, including trends, expectations about the future, and AI's impact on society. This section is a compilation of such comments and some selected predictions from experts.

AI is progressing rapidly, not only in terms of the quality of the products derived but also regarding the production process itself. As AI improves itself and more AI leads to even better AI, a continuous acceleration cycle emerges that cannot be stopped. However, apart from the usual warnings about how fast it is moving, there are urgent ethical predictions that demand attention. AI presents a fundamental challenge for consciousness and autonomy. It should not be left in the hands of a few corporations and wealthy individuals. It is the duty of humanity to steer AI toward the common good and keep it under collective control.

10.3. Long-term Implications for the Industry

Preparing for every test scenario has become impractical, and so long-term adaptation is imperative. Collaborating with other AI Alliance members to spot common blockers and identify optimization opportunities remains a top priority.

Machine Learning Testing is an emerging discipline, yet tenets of the field have practical relevance even at this preliminary stage. The present work examines AI-driven risks within QA and offers concrete solutions to mitigate them. Apart from the field's inherent challenges, Generative AI also brings inherent risks that require a mature, cautious, and ethical approach—including, but not limited to, ensuring the responsible use of AI capabilities, understanding the model's nature and suitability, preventing misuse, protecting intellectual property, and promoting the proper integration of AI into existing business processes.

The continual automation of tests and QA jobs leads to difficult questions. Recent studies estimate that the percentage of tasks susceptible to automation has climbed to 45%, up from 27% in 2016. Naturally, concerns are mounting regarding the long-term fate of QA personnel. The presented framework strives to eliminate or mitigate manual, repetitive, and monotonous work without rendering people obsolete.

11. Conclusion

Testing is a crucial part of ensuring that software meets its goals. But as software systems and the means of testing them evolve, new challenges arise. AI is a transformative tool for testing due to its ability to learn and find broad solutions. It's able to quickly generate many tests, quickly adapt to software changes, and in many cases generate tests that are more complete and nuanced than those produced by humans. However, this ability also

introduces unique challenges. AI systems have a wide range of implementation choices, complicated instruction processes, and can hand unexpected decisions back to the tester. This can make the testing itself hard to explain and difficult to predict.

Addressing these challenges requires a holistic understanding of the both the ways AI enhances testing and the ways it challenges it. The AI-Testing relationship presents ethical dimensions and influences the future of testing and the testers who perform it. By considering these aspects, one can create guidelines to help testers safely and effectively use AI-generated tests.

References

[1] Ramchand S, Shaikh S, Alam I. Role of artificial intelligence in software quality assurance. InProceedings of SAI Intelligent Systems Conference 2021 Aug 3 (pp. 125-136). Cham: Springer International Publishing.

[2] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.

[3] Layman L, Vetter R. Generative artificial intelligence and the future of software testing. Computer. 2024 Jan 3;57(1):27-32.

[4] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.

[5] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.

[6] Nguyen P, Shivadekar S, Chukkapalli SS, Halem M. Satellite data fusion of multiple observed XCO2 using compressive sensing and deep learning. InIGARSS 2020-2020 IEEE International Geoscience and Remote Sensing Symposium 2020 Sep 26 (pp. 2073-2076). IEEE.

[7] Aleti A. Software testing of generative ai systems: Challenges and opportunities. In2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) 2023 May 14 (pp. 4-14). IEEE.

[8] Xie T. The synergy of human and artificial intelligence in software engineering. In2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE) 2013 May 25 (pp. 4-6). IEEE.

[9] Vanmali M, Last M, Kandel A. Using a neural network in the software testing process. International Journal of Intelligent Systems. 2002 Jan;17(1):45-62.

[10] Bayrı V, Demirel E. Ai-powered software testing: The impact of large language models on testing methodologies. In2023 4th International Informatics and Software Engineering Conference (IISEC) 2023 Dec 21 (pp. 1-4). IEEE.

[11] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024 Feb 20;50(4):911-36.

[12] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[13] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.
[14] Lenz AR, Pozo A, Vergilio SR. Linking software testing results with a machine learning approach. Engineering Applications of Artificial Intelligence. 2013 May 1;26(5-6):1631-40.
[15] Rodríguez G, Soria Á, Campo M. Artificial intelligence in service-oriented software design. Engineering Applications of Artificial Intelligence. 2016 Aug 1;53:86-104.
[16] Alshahwan N, Harman M, Marginean A. Software testing research challenges: An industrial perspective. In2023 IEEE Conference on Software Testing, Verification and Validation (ICST) 2023 Apr 16 (pp. 1-10). IEEE.
[17] Panda SP. Mastering Microsoft Fabric Unified Data Engineering, Governance, and Artificial Intelligence in the Cloud. Governance, and Artificial Intelligence in the Cloud (January 22, 2025). 2025 Jan 22.
[18] Partridge D. Artificial intelligence and software engineering. Routledge; 2013 Apr 11.
[19] Bellamy RK, Dey K, Hind M, Hoffman SC, Houde S, Kannan K, Lohia P, Mehta S, Mojsilovic A, Nagar S, Ramamurthy KN. Think your artificial intelligence software is fair? Think again. IEEE Software. 2019 Jun 17;36(4):76-80.
[20] Panda SP. Enhancing Continuous Integration and Delivery Pipelines Using Azure DevOps and GitHub Actions. Available at SSRN 5285094. 2024 Jul 7.
[21] Rich C, Waters RC, editors. Readings in artificial intelligence and software engineering. Morgan Kaufmann; 2014 Jun 28.
[22] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.
[23] Zinchenko V, Chetverikov S, Akhmad E, Arzamasov K, Vladzymyrskyy A, Andreychenko A, Morozov S. Changes in software as a medical device based on artificial intelligence technologies. International Journal of Computer Assisted Radiology and Surgery. 2022 Oct;17(10):1969-77.
[24] Gurcan F, Dalveren GG, Cagiltay NE, Roman D, Soylu A. Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. IEEE Access. 2022 Oct 4;10:106093-109.
[25] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.
[26] Krichen M. How artificial intelligence can revolutionize software testing techniques. InInternational Conference on Innovations in Bio-Inspired Computing and Applications 2022 Dec 15 (pp. 189-198). Cham: Springer Nature Switzerland.
[27] Islam M, Khan F, Alam S, Hasan M. Artificial intelligence in software testing: A systematic review. InTENCON 2023-2023 IEEE Region 10 Conference (TENCON) 2023 Oct 31 (pp. 524-529). IEEE.
[28] Tahvili S, Hatvani L. Artificial intelligence methods for optimization of the software testing process: With practical examples and exercises. Academic Press; 2022 Jul 21.

[29] Awad A, Qutqut MH, Ahmed A, Al-Haj F, Almasalha F. Artificial Intelligence Role in Software Automation Testing. In2024 International Conference on Decision Aid Sciences and Applications (DASA) 2024 Dec 11 (pp. 1-6). IEEE.
[30] Marijan D, Gotlieb A. Software testing for machine learning. InProceedings of the AAAI Conference on Artificial Intelligence 2020 Apr 3 (Vol. 34, No. 09, pp. 13576-13582).
[31] Last M, Kandel A, Bunke H, editors. Artificial intelligence methods in software testing. World Scientific; 2004 Jun 3.
[32] Boukhlif M, Hanine M, Kharmoum N. A decade of intelligent software testing research: A bibliometric analysis. Electronics. 2023 May 5;12(9):2109.
[33] Li JJ, Ulrich A, Bai X, Bertolino A. Advances in test automation for software with special focus on artificial intelligence and machine learning. Software Quality Journal. 2020 Mar;28(1):245-8.
[34] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.
[35] Serna M E, Acevedo M E, Serna A A. Integration of properties of virtual reality, artificial neural networks, and artificial intelligence in the automation of software tests: A review. Journal of Software: Evolution and Process. 2019 Jul;31(7):e2159.
[36] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.
[37] Felderer M, Enoiu EP, Tahvili S. Artificial intelligence techniques in system testing. InOptimising the Software Development Process with Artificial Intelligence 2023 Jul 20 (pp. 221-240). Singapore: Springer Nature Singapore.