REVIEW ARTICLE

# Software Testing Evolution: Comparative Insights into Traditional and Emerging Practices

**Samia Akhtar**[1,*]

[1] Department of Computer Science, Virtual University of Pakistan, Lahore 54000, Pakistan

## Abstract

**Software testing is a fundamental pillar of software engineering which ensures that applications function correctly, meet user requirements, and remain reliable under different conditions. As software systems become more complex and the demand for faster development grows, testing strategies have evolved to meet new challenges. This paper aims to comprehensively compare traditional and modern software testing techniques to provide practitioners with a structured understanding of their evolution, strengths, limitations, and applicability. It covers classical methods such as unit testing, integration testing, system testing, acceptance testing and other testing types like black-box, white-box, and grey-box. Each method is analyzed based on its purpose, advantages, limitations, and best use cases. The paper also explores current testing trends including AI-augmented testing, continuous testing in DevOps, shift-left and shift-right testing, and large scale automated testing. It highlights the growing importance of testing in cloud-native and microservices-based environments. These modern practices are evaluated for their impact on software quality assurance, particularly in improving test coverage, fault detection, usability, and security. The survey also identifies challenges faced by testing teams, such as flaky tests, tool complexity, test data management, and AI explainability. Finally, the paper offers future directions including smarter automation and more accessible testing tools. This work serves as a useful guide for software engineers, testers, researchers, and QA professionals seeking to understand the evolving role of software testing and apply effective strategies in modern development environments.**

**Keywords**: software testing strategies, software quality assurance (SQA), black box and white box testing, AI-augmented testing, test automation.

## 1 Introduction

Software testing is a crucial part of the software development lifecycle (SDLC). It helps ensure that applications work as expected, meet user needs, and provide a good user experience [1]. As software systems continue to evolve—becoming more cloud-based, AI-powered, and user-focused—testing is more important than ever. It is used not only for verifying software functionality but also for

**Academic Editor:**
Usama Ahmed

**Submitted:** 11 July 2025
**Accepted:** 24 July 2025
**Published:** 19 August 2025

**Vol.** 1, **No.** 1, 2025.
10.62762/JSE.2025.246843

***Corresponding author:**
Samia Akhtar
samiaakhtar9898@gmail.com

**Citation**
Akhtar, S. (2025). Software Testing Evolution: Comparative Insights into Traditional and Emerging Practices. *ICCK Journal of Software Engineering*, 1(1), 46–62.

assessing security, performance, usability, and ease of maintenance. With software playing a major role in critical fields like healthcare, finance, and self-driving systems, the need for strong and scalable testing methods has grown significantly [2].

Traditionally, software testing has been divided into different levels and techniques, each serving a specific role. Common types include unit testing, integration testing, system testing, and acceptance testing. These help catch bugs at various stages of development. Testers have also used black-box, white-box, and grey-box approaches, depending on how much they know about the system's internal code. These traditional methods remain highly relevant and are widely used in both academic and industry settings [3–7]. However, modern development practices like Agile, DevOps, and CI/CD now require faster and more frequent software updates. This has put more pressure on teams to test and release high-quality software quickly. As a result, there is a growing shift toward automated and AI-assisted testing tools. These tools reduce manual effort, improve test coverage, and speed up the process of finding bugs [8].

One major trend is the use of AI and machine learning in testing. Recent advancements in AI-driven automation, shift-left testing, and continuous integration have significantly transformed modern software testing practices [9, 10]. Tools such as GitHub Copilot, ChatGPT, and smart automation platforms are being used to generate test cases, find errors, analyze logs, and even suggest fixes [11]. While these tools offer many benefits, they also raise concerns about trust, explainability, and reliance on good-quality data. Developers and testers are now working to find the right balance between human input and AI support to ensure systems are not only efficient but also reliable and ethical. The rise of microservices, container-based apps and cross-platform software has further complicated the testing process. Now, teams must also test for interoperability, scalability, resilience, and security [12]. To handle this growing complexity, many organizations are adopting continuous testing, automated regression testing, and test orchestration platforms to maintain quality throughout the software's lifecycle. Figure 1 shows the goals, trends and some key testing areas in software testing.

While several prior works have discussed individual testing methods or focused on either traditional or modern strategies, few have provided a side-by-side comparison that links classical approaches with today's agile, AI-driven, and cloud-based practices. Moreover, existing literature often lacks practical insights into tool usage, real-world challenges like flaky tests or AI explainability, and the evolving role of testing in continuous delivery environments. This paper addresses these gaps by offering a unified survey that bridges foundational concepts with current innovations, highlighting both opportunities and limitations.

This survey paper provides an overview of software testing strategies, covering both traditional and modern practices. It compares classical testing methods by explaining where they work best and what their strengths and weaknesses are. It also explores how new technologies—like AI-powered testing and continuous monitoring—are changing the way testing is done. By reviewing recent research, tools, and trends from recent years, the paper also points out key challenges and suggests future directions for research. This makes it a valuable resource for software engineers, testers, researchers, and decision-makers who want to understand how software testing is evolving and what it means for software quality. In summary, this paper connects traditional software testing knowledge with modern tools and methods, helping the reader gain both practical and theoretical insights into today's software testing landscape. The main contributions of this paper are as follows:

- This paper provides a comparative review of classical testing methods, including unit, integration, system, and acceptance testing, as well as black-box, white-box, and grey-box techniques.
- It presents an overview of modern testing trends such as AI-augmented testing, continuous testing, shift-left and shift-right practices, and testing in cloud-native environments.
- It includes tabular comparisons that clearly differentiate traditional and modern strategies based on scalability, automation, and feedback speed.
- The paper discusses widely used testing tools and their applicability across both classical and modern testing workflows.
- It identifies key challenges in current testing practices, such as flaky tests and AI explainability, and suggests potential directions for future improvement

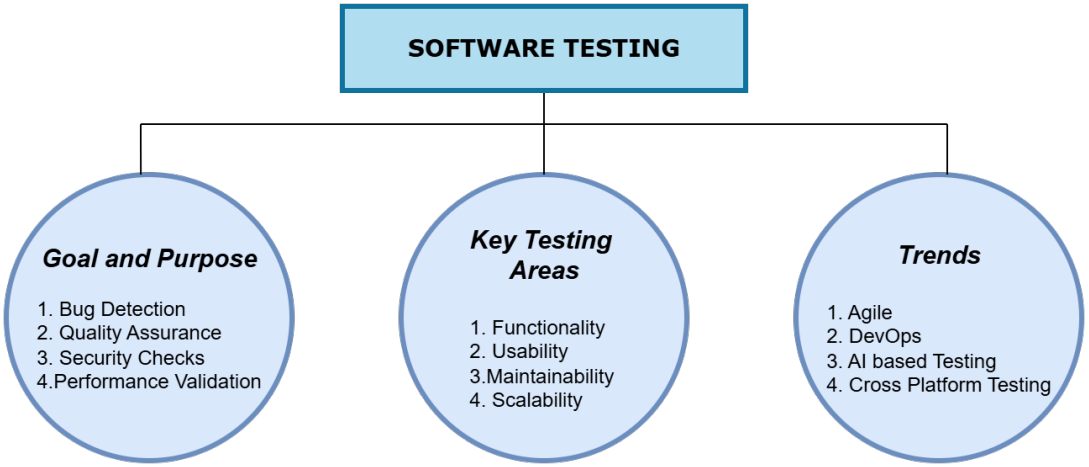


**Figure 1.** Overview of software testing highlighting its main goals, key focus areas, and current industry trends.

## 2 Related Work

The evolution of software testing has been well-documented through various scholarly works and industrial reports. In this section, we analyze relevant studies that provide a foundation for understanding current testing strategies and their impact on software quality assurance.

The authors in [13] have conducted a survey on prioritization in Automotive Software Testing. They investigated publication trends, commonly used prioritization methods, and the quality distribution of existing studies. Their goal was to assess how prioritization can reduce testing time while preserving reliability in safety-critical automotive systems. They emphasized that early fault detection is essential to avoid recalls and safety failures. The study provides insights for both practitioners and researchers, especially those adopting prioritization strategies. It also outlines research gaps, including a lack of studies on test case generation and selection. Future work includes conducting qualitative studies to better understand key testing activities in automotive contexts.

The authors in [14] have discussed how using artificial intelligence in test automation is changing the way software quality is managed. They explain that AI methods like machine learning and computer vision help reduce common problems such as fragile test scripts and high maintenance costs. The paper shows that AI can create tests automatically, fix broken scripts on its own, and find defects more accurately. It also describes a four-stage path from basic AI support to fully automatic testing systems. The authors highlight how AI improves performance testing by simulating real loads and spotting problems early. They point out that AI saves time, improves testing coverage, and speeds up development. In the end, the paper sees AI-based testing as a key tool for keeping software quality high in today's fast-moving projects.

In [15], researchers have studied how artificial intelligence and machine learning are helping improve different parts of software testing. They explain that traditional automation tools face problems in fast-changing CI/CD environments, and AI/ML can help solve these issues. The paper highlights key testing tasks such as bug detection, test maintenance, and automatic test generation where AI has shown strong results. It also points out which AI techniques are most commonly used in each activity. The researchers reviewed various AI-powered automation tools and how they support different testing needs. Their findings help researchers understand current developments and guide QA teams in choosing the right tools. They conclude that this knowledge can support better tool selection and lead to more efficient testing in modern projects.

In [16], researchers studied a new way to test deep neural networks (DNNs) using black-box input diversity metrics. They found that traditional white-box methods, which need access to a model's internal parts, are often hard to use and may not give reliable results. To solve this, they tested three simpler diversity measures on four datasets and five DNN models. Their results showed that one measure called geometric diversity was better at finding faults and faster than older methods. The authors also used a clustering method to group similar mistakes and estimate faults more accurately. Their approach does not need to run the model or rely on its output, which

makes it useful in many real-world situations. They suggest using geometric diversity as a helpful and practical way to guide DNN testing.

Researchers explored how artificial intelligence can improve the software testing process by making it faster, smarter, and more accurate in [17]. They highlight the need for quicker testing due to growing software complexity and business demands. The paper discusses key AI pillars like machine learning, deep learning, and natural language processing that support better test automation. It suggests that AI-driven testing will become central to quality assurance, helping organizations release higher-quality software faster. The authors also predict that AI will work closely with future technologies like cloud computing, IoT, and big data. They emphasize that testers will shift roles, focusing more on training and fine-tuning AI models. Overall, the paper sees AI as a major driver of smarter, more efficient, and fully automated testing in the future.

Researchers in [18] studied how adding automated testing early in the CI/CD pipeline can improve software quality and speed up development. They focused on the shift-left approach, which means finding and fixing problems as early as possible to save time and reduce costs. The paper explains different types of tests—like unit, integration, and end-to-end tests—and how they fit into early testing. It shows how shift-left changes the usual testing process by starting tests during the design phase instead of later. The authors use real examples to show that this method leads to better software, faster development, and lower expenses. They also suggest ways to handle common problems like test setup, data management, and keeping tests up to date. In the end, they found that using automated testing with the shift-left approach helps teams catch bugs sooner and deliver better software more quickly.

In [19], researchers emphasized the importance of software testing in improving quality, reducing bugs, and lowering development costs. They explain that testing is a continuous process meant to find defects rather than prove perfection. The paper reviews common techniques such as white-box, black-box, grey-box, and security testing, showing how each contributes to checking different parts of a system. The authors point out that testing is time-consuming and requires proper planning, which is why automation and modern tools are becoming essential. They stress the need for testers to understand core testing principles and suggest that future testing will rely more on simulation, automation, and model-based approaches for better results.

In [20], authors reviewed automatic test data generation, mainly used in compiler testing, and discussed its potential for broader software applications. They outlined key techniques, benefits, and limitations of using it alongside other testing methods. In [21], researchers explored AI-based testing, showing how machine learning and NLP improve test case generation and validation. They highlighted gains in efficiency and accuracy, while noting challenges like data quality, AI bias, and integration with CI/CD workflows.

In summary, prior studies have focused on specific areas like AI-based automation, prioritization, or early testing in CI/CD pipelines. However, they often lack a unified comparison of classical and modern strategies, as well as practical insights into tools and real-world challenges like flaky tests and AI explainability. This paper addresses these gaps through a side-by-side analysis, tabular comparisons, and actionable insights relevant to today's development practices.

Compared to the work of Patel [14] and Pham et al. [15] for instance, which focus on the role of AI in testing automation, our paper provides a broader perspective by not only covering AI-based techniques but also integrating them with classical methods like unit, system, and acceptance testing. While those studies concentrate on automation and intelligent tools, we expand the discussion to include practical comparisons, modern challenges, and how these methods coexist in real-world development environments.

## 3 Methodology

The methodology for this survey was designed to offer a structured and clear overview of software testing strategies, covering both traditional techniques and recent advancements in modern software engineering. The aim is to explore the evolution of testing practices, highlight current trends, and understand their impact on software quality assurance. The survey follows a qualitative approach, focusing on core testing concepts and their practical relevance in different development environments.

### 3.1 Scope of the Survey

This survey includes widely used software testing techniques that are directly related to improving

software quality. The scope covers classical methods such as unit testing, integration testing, system testing, and acceptance testing, along with key testing types like black-box, white-box, and grey-box approaches. It also considers recent trends in the field, including AI-assisted testing, test automation, and continuous testing practices commonly used in Agile and DevOps environments. This focus on both classical and modern techniques was chosen to give a complete picture of how software testing has changed over time, helping readers understand both old methods and new trends in real-world practice.

### 3.2 Review Approach

A variety of relevant research papers, technical documents, and domain-relevant literature were reviewed to gather insights into the evolution and application of testing techniques. The focus was on works that present practical strategies, comparative discussions, or conceptual frameworks related to both established and emerging testing methods. Special attention was given to studies and discussions from recent years to ensure that modern trends and challenges are well represented.

### 3.3 Selection Criteria

The materials selected for review were chosen based on the following criteria:

- Relevance to software testing practices, tools, or quality assurance strategies
- Inclusion of classical or modern testing methods with practical implications
- Clear explanation of concepts, benefits, limitations, or comparisons
- Alignment with contemporary software development models and needs

### 3.4 Analytical Approach

The reviewed content was analyzed qualitatively to extract common testing objectives, strengths and weaknesses of each method, and their use in real-world scenarios. Instead of relying on numerical metrics or empirical results, the analysis focused on conceptual clarity and practical relevance. Key techniques were compared based on factors such as:

- Level of abstraction (e.g., unit-level vs. system-level testing)
- Tester access to internal code or structure
- Suitability for automation
- Applicability in Agile or DevOps workflows
- Overall strengths and trade-offs

### 3.5 Traditional Trends in Software Testing

Before the rise of modern development models like Agile and DevOps, software testing followed more structured and sequential practices [22]. These traditional testing trends laid the groundwork for many current quality assurance strategies. This section explores the foundational testing methods and approaches that have been widely adopted over the years, such as unit testing, integration testing, system testing, and classical black-box and white-box techniques. All the approaches to be discussed in this section are summed up in Figure 2.

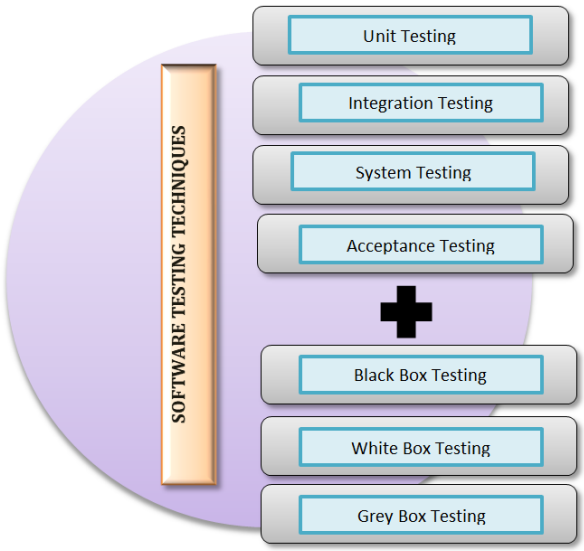


**Figure 2.** Some traditional software testing strategies.

#### 3.5.1 *Unit Testing*

Unit testing focuses on verifying the smallest testable components of a software system, such as functions, methods, classes, or modules [23]. The goal is to ensure each unit behaves as expected according to its design and specifications. Developers typically write and execute unit tests during the development phase, either manually or using automation tools. Each method or feature is tested individually to detect and fix errors early. Common techniques in unit testing include module interface assessment (to check input and output consistency), validation of temporary data structures, testing boundary conditions, covering all independent paths, and verifying error-handling routes to ensure the module responds properly to

unexpected inputs or failures. Table 1 provides the benefits and challenges of using unit testing.

### 3.5.2 Integration Testing

Integration testing focuses on verifying the interactions between different software modules once they are combined [24]. After individual units are tested, this phase ensures that the components work together correctly and that data flows smoothly across interfaces. It is especially useful for detecting communication mismatches, interface issues, and unexpected interactions that may not surface during unit testing. Integration testing can be carried out using various strategies, including top-down, bottom-up, and hybrid approaches, depending on the system structure [25]. It also plays a key role in validating the system's architecture, internal workflows, and integration with external components or APIs. Overall, integration testing acts as a bridge between unit testing and full system testing in the development lifecycle. Pros and Cons of this testing strategy are given in Table 2.

### 3.5.3 System Testing

System testing is a high-level testing phase where the complete and integrated software is tested as a whole [26]. It focuses on verifying that the system meets its functional and non-functional requirements as specified during the design phase. This type of testing simulates real-world usage and checks how the software behaves under different conditions, including stress, performance, security, and compatibility. System testing is typically performed by a separate quality assurance team and not by the developers. It helps ensure that the software behaves consistently and reliably across various environments [27]. By covering end-to-end scenarios, system testing provides confidence that the product is ready for deployment and meets both user expectations and business objectives. Table 3 provides the benefits and challenges for this technique.

### 3.5.4 Acceptance Testing

Acceptance testing is the final phase of software testing, carried out to ensure that the developed system meets business requirements and is ready for deployment [28]. It is usually performed by the end users, clients, or a testing team representing the customer. This type of testing focuses on validating the software from the user's perspective, checking whether the system performs as expected in real-world scenarios. Acceptance testing can be formal (User Acceptance Testing – UAT) or informal, and it may include both functional and non-functional checks. Its main goal is to gain confidence that the software is complete, reliable, and usable. A successful acceptance test signifies that the product is fit for release and operational use [29]. Table 4 presents the pros and cons for this strategy.

### 3.5.5 Black-Box Testing

Black-box testing is a software testing technique where the internal structure or code of the system is not known to the tester [30–32]. Instead, the focus is on examining inputs and expected outputs to verify whether the software behaves correctly. Testers validate functional requirements without considering how the functionality is implemented. This method is widely used for system, acceptance, and regression testing. It helps ensure that the application meets user expectations and behaves correctly across various inputs and user actions. Black-box testing can be applied to all levels of testing but is most common at the system and acceptance levels. It is especially useful when developers and testers work independently to avoid bias or assumptions in validation. Advantages and disadvantages of this technique are in Table 5.

### 3.5.6 White-Box Testing

White-box testing, also known as structural or clear-box testing, involves examining the internal structure, logic, and code of a software application [33]. Unlike black-box testing, this technique requires the tester to have full knowledge of the source code, making it most suitable for developers or technically skilled testers. The goal is to test individual paths, conditions, loops, and branches within the code to ensure they function as expected. White-box testing is commonly applied at the unit and integration levels, but it can also be used to assess security, performance, and logic flaws. It provides detailed insight into the software's internal workings and helps uncover hidden errors that may not be detected through external testing alone [34]. Table 6 provides the benefits and challenges of using white-box testing.

### 3.5.7 Grey-Box Testing

Grey-box testing combines elements of both black-box and white-box testing. In this approach, testers have partial knowledge of the system's internal workings—such as access to architecture diagrams, database schemas, or high-level code logic—while still focusing on input-output behavior [35]. This hybrid method is useful for validating both the internal flow and external functionality of an application. It enables testers to design better-informed test cases

**Table 1.** Pros and Cons of Unit Testing.

| Advantages | Disadvantages |
|---|---|
| Promotes high-quality and well-structured code. | Can be time-consuming to implement thoroughly. |
| Cost-effective in detecting and fixing early-stage bugs. | Writing effective and meaningful test cases can be challenging. |
| Allows isolated testing of specific components without relying on other parts of the system. | May not detect integration-level issues since units are tested independently. |
| Simplifies fault detection and reduces debugging effort. | Integration bugs or system-level errors may go unnoticed. |
| Focuses on small code segments, making the testing process simple and targeted. | Errors made by the developer can affect the entire system if not caught early. |

**Table 2.** Pros and Cons of Integration Testing.

| Advantages | Disadvantages |
|---|---|
| Ensures that different modules or components work together correctly. | Can be complex to manage due to dependencies between components. |
| Helps identify interface mismatches and communication issues between modules. | Test case design becomes increasingly difficult as the number of integrated components increases. |
| Catches system-level issues that unit testing may miss. | Errors in individual modules can impact the overall test results, making fault isolation harder. |
| Supports early detection of architectural or data flow problems. | Requires detailed planning and may demand the presence of partially developed systems (stubs/drivers). |
| Improves overall system reliability by validating component interactions. | Maintenance of integration tests can be time-consuming, especially in evolving systems. |

**Table 3.** Pros and Cons of System Testing.

| Advantages | Disadvantages |
|---|---|
| Validates the complete software system against requirements. | Requires significant time and resources to execute fully. |
| Detects issues missed during unit and integration testing. | Difficult to trace the exact source of failure in complex systems. |
| Covers both functional and non-functional aspects like performance and security. | Test case design is complex due to wide scope. |
| Ensures system behaves correctly in real-world conditions. | Can be costly, especially for large-scale systems. |
| Enhances overall product quality and user confidence. | May require complex test environments and setup. |

and uncover defects that are missed in pure black-box or white-box testing. Grey-box testing is especially beneficial in integration and system-level testing, where understanding internal structures can improve efficiency without needing full code access. It balances test depth and realism, making it valuable in complex, layered systems involving APIs, databases, or middleware [36]. Table 7 provides the benefits and challenges of using grey-box testing.

### 3.6 Modern Trends in Software Testing

In recent years, software testing has undergone a substantial transformation due to evolving software development practices, increasing system complexity, and the push for rapid, high-quality releases [37]. While foundational testing strategies remain essential, modern approaches now reflect a shift toward automation, intelligence, and tight integration with DevOps and AI-driven workflows. The following

**Table 4.** Pros and Cons of Acceptance Testing.

| Advantages | Disadvantages |
|---|---|
| Validates the system from the user's point of view. | Often dependent on user availability and cooperation. |
| Ensures the software meets business goals and requirements. | May miss technical or backend issues due to user focus. |
| Builds customer confidence before the product is launched. | Users may lack technical expertise to conduct thorough testing. |
| Acts as the final approval before deployment. | Feedback can be subjective and vary between testers. |
| Helps identify usability and functionality issues in real scenarios. | Test cases may not cover all possible edge cases or exceptions. |

**Table 5.** Pros and Cons of Black-Box Testing.

| Advantages | Disadvantages |
|---|---|
| Tests the system from the user's perspective without needing to understand the internal code. | Limited ability to identify internal code or logic errors. |
| Useful for validating functional requirements and user interactions. | Can miss logical errors if not all input paths are tested. |
| Effective for large systems and complex user interfaces. | Test case design relies heavily on requirements and documentation. |
| Allows independent testing by non-developers. | Difficult to trace the cause of failure due to lack of internal visibility. |
| Encourages unbiased testing since implementation details are hidden. | Inefficient for testing edge cases or internal boundaries. |

**Table 6.** Pros and Cons of White-Box Testing.

| Advantages | Disadvantages |
|---|---|
| Offers in-depth testing by covering internal logic and control structures. | Requires detailed knowledge of the source code, limiting who can perform it. |
| Helps detect hidden errors, unreachable code, and security vulnerabilities. | Not suitable for validating user-facing features or UI behavior. |
| Supports high test coverage with path, loop, and condition testing. | Time-consuming, especially for large and complex codebases. |
| Aids in code optimization and improving overall code quality. | Maintenance becomes harder if code changes frequently. |
| Useful for early-stage testing, especially during development. | May miss integration or system-level issues not visible at code level. |

sub-sections discuss the key trends that have shaped the software testing landscape. These trends are summed up in Figure 3.

### *3.6.1 AI-Augmented Testing*

AI-augmented testing refers to the integration of artificial intelligence techniques—such as machine learning, natural language processing, and pattern recognition—into the software testing process [38–40]. Unlike traditional automation, which depends on static, human-written test scripts, AI-powered tools are capable of learning from data, adapting to application changes, and making predictions about potential problem areas. These tools can analyze source code, user stories, or even documentation written in natural language to automatically generate relevant test cases. They can also detect changes in the user interface and autonomously update or "self-heal" test scripts, reducing manual intervention. Intelligent visual testing is another powerful feature of AI-driven tools, enabling them to identify layout

**Table 7.** Pros and Cons of Grey-Box Testing.

| Advantages | Disadvantages |
|---|---|
| Combines the strengths of both black-box and white-box testing. | Partial knowledge may still limit full code-level analysis. |
| Allows more accurate and focused test case design. | Requires testers with both technical and functional knowledge. |
| Helps detect issues in data flow, integration points, and internal processes. | May not fully explore deep logic errors without full code access. |
| Suitable for testing layered systems like web apps, APIs, and databases. | Defining boundaries between what's tested externally vs. internally can be tricky. |
| Increases test coverage without requiring full developer-level access. | Can become complex in large systems with many interdependencies. |

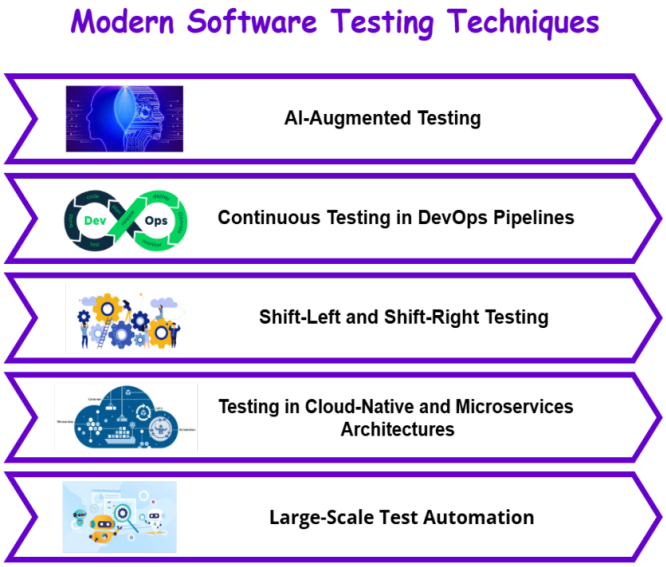


**Figure 3.** Modern software testing strategies discussed in this study.

issues that might otherwise go unnoticed. Platforms such as Testim, Applitools, and Functionize are already implementing these capabilities in real-world QA environments.

**Benefits and Challenges** AI-augmented testing brings several advantages that enhance the speed, accuracy, and adaptability of software testing. It reduces manual effort by generating and maintaining test cases automatically, increases test coverage by exploring diverse inputs, and supports continuous testing in fast-paced development cycles. AI tools can detect UI changes and self-heal test scripts, improving test stability, while intelligent visual testing helps identify subtle layout issues. Additionally, AI's ability to learn from past defects enables early bug detection. However, challenges remain. AI-generated results can be difficult to interpret, especially in high-risk systems where transparency is critical. Trust in AI decisions is still developing, and the quality of outcomes depends heavily on the data used to train the models. Issues like overfitting, false positives, and poor generalization to new environments require human oversight and careful tuning to ensure consistent and reliable testing performance.

### 3.6.2 *Continuous Testing in DevOps Pipelines*

Continuous testing is a modern testing approach designed to support Agile and DevOps workflows [41]. It involves running automated tests consistently throughout the entire software development lifecycle, especially within CI/CD (Continuous Integration and Continuous Delivery) pipelines. Instead of waiting until the end of development to test, continuous testing integrates quality checks at every stage—from code commit to deployment. This ensures that defects are caught early, feedback is provided quickly, and code remains production-ready at all times. Common tools like Jenkins, GitLab CI, GitHub Actions, and CircleCI help trigger and manage automated test suites, including unit, integration, and end-to-end tests, every time new code is added. As a result, development teams can identify and resolve issues before they reach production, maintain higher software quality, and deliver updates more reliably and efficiently.

**Benefits and Challenges** Continuous testing offers several important advantages. It accelerates the feedback cycle, reduces post-deployment failures, and increases the speed of software delivery. By testing every code change automatically, it minimizes risks and ensures consistent software quality. This approach also encourages better collaboration among developers, testers, and operations teams, who now share responsibility for quality. Moreover, it supports faster decision-making and improves

confidence in each build before release. Despite its strengths, continuous testing brings specific challenges. Managing test data and maintaining stable test environments can be difficult, especially in systems with many services or configurations [42]. Flaky tests that pass or fail unpredictably may reduce trust in test results. Additionally, achieving a balance between test coverage and pipeline performance requires careful optimization. Poorly designed tests or long execution times can slow down development. Still, when implemented thoughtfully, continuous testing is essential for achieving both speed and reliability in modern software delivery.

#### *3.6.3 Shift-Left and Shift-Right Testing*

Shift-left and shift-right testing are two complementary strategies used to improve software quality across the development lifecycle [43]. Shift-left testing moves testing earlier in the process during planning, design, and coding. Techniques like Test-Driven Development (TDD) and Behavior-Driven Development (BDD) help developers write tests early, detect bugs sooner, and produce cleaner, more reliable code. This approach encourages collaboration and helps reduce the cost and effort of fixing issues later. Shift-right testing takes place after deployment, using real-world data to monitor performance and user behavior. Methods like A/B testing, canary releases, and chaos engineering help teams identify problems in production and fine-tune future releases. Together, these approaches ensure that quality is built in early and continuously validated in real use.

**Benefits and Challenges** Shift-left testing reduces development delays by catching bugs early and improving code quality. It promotes collaboration between developers and testers and aligns testing with requirements. Shift-right testing adds real-time insights, helping teams detect unexpected issues in production and refine user experience. Challenges exist for both. Shift-left requires teams to adopt early testing habits and automation tools. Shift-right depends on strong monitoring and safe release practices. Implementing both effectively takes coordination and maturity, but together they lead to better, faster, and more resilient software.

#### *3.6.4 Testing in Cloud-Native and Microservices Architectures*

Cloud-native development and microservices architecture have reshaped how modern software is built and tested. Unlike traditional monolithic applications, microservices split functionality into small, independent services that interact through APIs. This design improves scalability and agility but introduces new testing challenges due to the distributed, dynamic nature of such systems. Testing must now cover not only individual services but also how they work together [44]. Unit and integration tests remain important but are not enough. Microservices need to be tested both in isolation and in combination with others. Contract testing helps verify communication between services even when they're developed separately. Tools like Pact and service virtualization make this possible by simulating dependencies. Since cloud-native applications run in containers using platforms like Docker and Kubernetes, test environments must closely match production settings to ensure realistic and reliable testing outcomes.

**Benefits and Challenges** Testing in cloud-native systems improves fault detection, speeds up feedback, and supports scalable development. Cloud platforms also offer tools for automation, monitoring, and load testing. However, coordinating tests across many services is difficult, especially with different teams or technologies. Managing test data, handling service versions, and debugging across distributed components are ongoing challenges. Still, with the right strategies, testing in microservices and cloud environments ensures better performance and reliability.

#### *3.6.5 Large-Scale Test Automation*

Large-scale test automation has become essential in modern software development, especially with the rise of faster release cycles and growing codebases. It involves using automation tools to execute tests—such as unit, integration, functional, and performance tests—across various environments without manual input. Unlike traditional manual testing, which is slow and prone to human error, large-scale automation allows teams to run thousands of tests in parallel within CI/CD pipelines [45]. This ensures quick feedback, reduced regression time, and better software stability. Tools like Selenium, Cypress, Playwright, Postman, and JMeter have made it easier to automate both UI and backend testing. Cloud integration further enhances scalability by allowing tests to run on virtual environments and containers, making automation more accessible and efficient.

**Benefits and Challenges** Test automation at scale improves productivity, enables faster releases, and increases test coverage [46]. It supports

early bug detection and maintains quality, even as systems grow more complex. Tools with low-code or record-and-playback features also allow non-developers to contribute, making automation more inclusive. However, maintaining stable test scripts is challenging, especially with frequent UI changes or data issues. Managing large test suites, version compatibility, and execution time requires careful planning. Automation may also miss usability flaws or creative bugs that require human insight. Without test optimization or proper monitoring, build times can slow down instead of speeding up. Still, when applied thoughtfully, large-scale automation becomes a key driver of agile success, continuous testing, and long-term software reliability.

## 4 Comparison and Discussion

This section discusses and compares the different software testing techniques explored in the paper. It reflects on both traditional and modern methods, highlighting how each one fits into today's software development practices. By examining key aspects such as test coverage, ease of automation, technical skills required, and how well each method works in real projects, we can better understand their strengths and limitations. Traditional techniques like unit testing, integration testing, and black-box testing have long been essential in software quality assurance. They are well-suited for structured environments and help ensure that each part of the system works correctly. These methods are clear, reliable, and often easier to apply when the system is stable and predictable. However, in fast-moving projects where software is released frequently, these traditional methods can fall short, especially when it comes to speed, scalability, and handling change.

Modern testing trends—such as AI-augmented testing, shift-left and shift-right practices, continuous testing in DevOps, and large-scale test automation—help fill these gaps. They are designed to support rapid development and constant feedback. For example, AI tools can create and update test cases automatically, while continuous testing checks the system at every stage of development. These practices are highly useful in agile, cloud-based, and distributed systems where fast feedback and real-time quality checks are critical. Still, modern techniques come with their own challenges. Setting up large-scale automation or AI-powered testing often requires more time, better tools, and experienced testers. Tests may fail due to minor changes in the system or unstable environments. Similarly, shift-right testing—done after release—needs strong monitoring and the ability to safely test in production. Without proper planning, these advanced methods can become difficult to manage and may slow down development instead of helping it. Table 8 presents the comparative overview of all the discussed strategies. Table 9 highlights the major differences between traditional and modern testing strategies.

One clear takeaway is that no single testing method works best for all situations. Traditional methods are dependable and straightforward, while modern techniques offer flexibility, speed, and deeper insights. The most effective approach is usually a combination of both. By using classical testing for stability and structure, and modern testing for agility and coverage, teams can build a more complete and reliable testing process. In practice, this means applying the right mix based on the project's goals, team skills, and system complexity. Combining unit and integration tests with automation, AI, and post-release monitoring can lead to better results. However, this also requires good coordination between developers, testers, and operations teams, as well as the right tools and planning. In conclusion, testing is no longer a one-time step in development—it is a continuous activity that must evolve with the software itself. The reported automation levels, scalability ratings, and tool support shown in our comparison tables help practitioners assess how feasible or impactful each technique is in real-world settings. High automation support, for instance, implies faster feedback and reduced manual workload, while scalability reflects how well a method handles growing system complexity. By understanding when and how to apply each method, teams can deliver higher-quality products, reduce risks, and keep up with the demands of modern software development.

To highlight how this survey contributes beyond existing studies, Table 10 has been added below. It contrasts this work with selected recent papers that focus on specific areas of software testing. Unlike prior studies that concentrate solely on modern trends or classical methods, this survey provides a broader perspective. The comparison also considers tool insights, structure, and practical value.

### 4.1 Impact of Testing Strategies on Software Quality Assurance

Software Quality Assurance (SQA) is essential for ensuring that software meets expected standards in

**Table 8.** Comparative Overview of Classical and Modern Software Testing Strategies Across Key Evaluation Criteria.

| Strategy | Level of Testing | Test Focus | Automation Support | Scalability | Primary Benefits | Key Challenges | Common Tools |
|---|---|---|---|---|---|---|---|
| Unit Testing | Unit | Code logic | High | High | Early bug detection, supports TDD | Limited to isolated code, may miss integration issues | JUnit, NUnit, PyTest, GoogleTest |
| Integration Testing | Integration | Module interaction | Medium | Medium | Verifies data flow and component interaction | Complex debugging, relies on stable components | TestNG, Postman, SoapUI |
| System Testing | System | Full system behavior | Medium | High | Validates overall system functionality | Time-consuming, may miss lower-level errors | Selenium, Katalon, TestComplete |
| Acceptance Testing | Acceptance | Business requirements, usability | Low | Medium | Ensures product meets user expectations | Subjective feedback, late discovery of critical issues | Cucumber, Zephyr, FitNesse |
| Black-Box Testing | All levels | Functional behavior | Medium | High | Tests from user's perspective, requires no code knowledge | Misses internal logic errors, may need large test sets | TestLink, Selenium, Postman |
| White-Box Testing | Unit /System | Internal code paths | High | Medium | High code coverage, logical validation | Requires code access, less user-focused | JaCoCo, JUnit, PyTest |
| Grey-Box Testing | Integration /System | Partial logic + behavior | Medium | Medium | Balanced view of logic and behavior | Limited access to internals, relies on design artifacts | Selenium, SoapUI |
| AI-Augmented Testing | UI /API /System | Predictive test generation, visuals | High | High | Intelligent test generation, adaptive scripts | Requires quality data, explainability limitations | Applitools, Testim, Mabl |
| Continuous Testing (DevOps) | All stages | End-to-end, regression, feedback | High | High | Real-time defect detection, integrates with CI /CD | Environment stability, flaky tests | Jenkins, GitHub Actions, GitLab CI |
| Shift-Left Testing | Unit /Integration | Early-phase logic, requirement validation | High | High | Prevents early defects, improves design alignment | Requires cultural shift and technical setup | Cucumber, SonarQube, ESLint |
| Shift-Right Testing | Production | Performance, monitoring, feedback | Medium | High | Captures real-world behavior, validates user experience | Needs observability, may expose users to failures | New Relic, Dynatrace, Gremlin |
| Cloud-Native & Microservices Testing | Integration /System | Service communication, scaling | High | High | Simulates real-world deployment, supports fault isolation | Complex setup, inter-service dependencies | Docker, Pact, Kubernetes |
| Large-Scale Test Automation | All levels | Broad test coverage | High | High | Fast feedback loops, reduced regression effort | Maintenance burden, brittle scripts | Selenium, Cypress, Playwright, Robot Framework |

**Table 9.** Comparative Summary of Foundational Differences Between Classical and Modern Software Testing Approaches.

| Aspect | Classical Testing Strategies | Modern Testing Strategies |
|---|---|---|
| Testing Philosophy | Sequential and phase-based | Integrated, continuous, and iterative |
| Development Model Alignment | Waterfall, V-Model, structured SDLC | Agile, DevOps, CI/CD |
| Focus Area | Functional correctness, requirement validation | Speed, adaptability, automation, real-time feedback |
| Timing of Testing | Mid to late development phase | Throughout the lifecycle (early and post-deployment) |
| Manual vs. Automated | Mostly manual or semi-automated | Heavily automated using smart tools |
| Tester Role | Dedicated QA professionals | Developers, testers, and DevOps engineers share responsibility |
| Tool Dependence | Mature but often standalone tools | Integrated toolchains and cloud-native platforms |
| Environment | Static test environments | Dynamic, containerized, and cloud-based environments |
| Scalability | Limited scalability in large systems | High scalability for distributed and large-scale systems |
| Test Coverage | Focused on defined features and paths | Broader scope including edge cases, behavioral patterns, user experience |
| Test Feedback Speed | Slower feedback cycles | Fast and continuous feedback |
| Flexibility | Less adaptable to changing requirements | Highly flexible and responsive to change |
| Example Techniques | Unit testing, Integration testing, System testing, Acceptance testing | AI-Augmented testing, Continuous testing, Shift-left/right, Cloud-native |
| Challenges | Late defect detection, manual overhead, slower release cycles | Tool complexity, flaky tests, reliance on automation, learning curve |
| Overall Objective | Validate functionality and requirement compliance | Ensure rapid, reliable, and resilient delivery at scale |

functionality, performance, reliability, security, and user satisfaction [45]. Testing plays a central role in SQA by identifying bugs, weaknesses, and risks that could affect the final product. Classical testing strategies—such as unit, integration, system, and acceptance testing—provide a structured approach to verify software components at different levels and are particularly valuable in long-term or safety-critical projects. However, they are typically applied later in the development cycle, which can lead to delays and higher costs if issues are discovered late. In contrast, modern testing strategies like continuous testing, AI-augmented testing, shift-left, and shift-right approaches emphasize early, frequent, and automated testing, enabling faster feedback, better scalability, and improved user experience. These methods also support real-time monitoring, usability improvements, and early risk detection. Still, modern practices face challenges such as test maintenance, flaky test behavior, trust in AI decisions, and managing complex environments. In practice, many teams adopt a hybrid approach—starting with classical tests and incorporating automation, monitoring, and user feedback—to maintain quality across the entire software lifecycle. Ultimately, selecting the right mix of testing strategies is crucial for building reliable, secure, and adaptable software that meets evolving user and business needs.

### 4.2 Challenges and Future Directions

Despite advancements in software testing, several challenges continue to impact modern development. Automated test maintenance is a major concern, as frequent UI or code changes often break scripts—especially in continuous testing setups. Test data management is also complex, particularly in

**Table 10.** Comparison of this survey with selected prior works.

| Study | Focus Area | Classical Techniques Covered | Modern Testing Trends Covered | Tool/Practical Insights | Comparative Discussion | Unique Contribution |
|---|---|---|---|---|---|---|
| [14] | AI in Software Testing | No | Yes | Partial | No | Focus on AI-based testing tools and methods |
| [15] | CI/CD + Test Automation | No | Yes | Yes | No | Explores continuous testing and pipelines |
| [16] | Deep Learning Test Coverage | No | Yes | No | No | Introduces test diversity metric |
| [19] | Classical Testing Overview | Yes | No | No | No | Describes foundational testing only |
| This Survey | Classical + Modern Testing | Yes | Yes | Yes | Yes | Unified, structured view with practical tools |

distributed and cloud-native systems where services must stay synchronized. Flaky tests reduce trust in results and complicate debugging. AI-driven testing introduces power but also risk, as many tools lack transparency and rely on high-quality training data. Additionally, the growing need for strong technical skills in testing creates barriers for non-expert QA professionals. These issues can slow down development and compromise software quality if not properly addressed.

To address these challenges, the future of testing should focus on smarter, more adaptable solutions. Tools powered by explainable AI would increase trust and accountability, particularly in regulated domains. Advancing self-healing test scripts and automated data generation can reduce manual effort and improve test reliability. Testing in production—using feature flags, canary releases, and real-time monitoring—can help teams gather valuable feedback from real usage. Low-code and no-code platforms should be expanded to make automation accessible to a wider range of users. Moving forward, ethical and inclusive testing practices, combined with closer collaboration among developers, testers, and researchers, will be essential to building reliable and resilient software systems.

## 5 Conclusion

Software testing remains a critical part of software engineering, playing a central role in ensuring that software is functional, reliable, secure, and aligned with user needs. This paper has presented a comprehensive survey of classical and modern testing strategies, comparing their principles, applications, strengths, and limitations. Classical approaches—such as unit testing, integration testing, system testing, and acceptance testing—continue to provide a strong foundation for structured quality assurance. At the same time, modern trends like AI-augmented testing, continuous testing in DevOps pipelines, shift-left and shift-right practices, and cloud-native testing have introduced more dynamic and scalable solutions. While modern methods offer clear benefits, they also come with challenges such as flaky tests, tool complexity, the need for technical upskilling, and concerns over AI explainability. Nevertheless, the integration of classical discipline with modern agility presents a promising path forward. A hybrid testing approach—tailored to the project's size, architecture, and team expertise—can help organizations build more robust, maintainable, and user-friendly software. In conclusion, testing is no longer a standalone phase but an ongoing process that supports software quality across every stage of development and deployment. As technologies continue to evolve, testing strategies must also adapt automation, intelligence, and user feedback.

The comparative framework in this study offers clear and organized insights into both traditional and modern testing techniques. It helps readers understand how different approaches apply in real-world development. One of its strengths is balancing well-known practices with newer trends in a practical way. However, since the study is

based on literature, it doesn't include hands-on experiments or tool-based testing results. This means real-life outcomes might differ based on the specific tools, teams, or environments used. Future work could explore these strategies through case studies or practical implementation to strengthen the findings. Future work could also focus on making testing smarter, more transparent, and more inclusive, enabling teams to deliver high-quality software that meets the needs of both businesses and users.

## Data Availability Statement

Data will be made available on request.

## Funding

This work was supported without any funding.

## Conflicts of Interest

The authors declare no conflicts of interest.

## Ethical Approval and Consent to Participate

Not applicable.

## References

[1] Kumar, S. (2023). Reviewing software testing models and optimization techniques: an analysis of efficiency and advancement needs. *Journal of Computers, Mechanical and Management, 2*(1), 32-46. [Crossref]

[2] Pargaonkar, S. (2023). A study on the benefits and limitations of software testing principles and techniques: software quality engineering. [Crossref]

[3] Khaliq, Z., Farooq, S. U., & Khan, D. A. (2022). Artificial intelligence in software testing: Impact, problems, challenges and prospect. *arXiv preprint arXiv:2201.05371*. [Crossref]

[4] Nama, P., Bhoyar, M., & Chinta, S. (2024). Autonomous test oracles: integrating ai for intelligent decision-making in automated software testing. *Well Testing Journal, 33*(S2), 326-353.

[5] Hunko, I. (2025). Adaptive Approaches to Software Testing with Embedded Artificial Intelligence in Dynamic Environments. *International Journal of Current Science Research and Review, 8*(05). [Crossref]

[6] Najihi, S., Elhadi, S., Ait Abdelouahid, R., & Marzak, A. (2022). Software Testing from an Agile and Traditional view. *Procedia Computer Science, 203*, 775-782. [Crossref]

[7] Formica, F., Fan, T., & Menghi, C. (2023). Search-based software testing driven by automatically generated and manually defined fitness functions. *ACM Transactions on Software Engineering and Methodology, 33*(2), 1-37. [Crossref]

[8] Delgado-Pérez, P., Medina-Bulo, I., Álvarez-García, M. Á., & Valle-Gómez, K. J. (2021, May). Mutation testing and self/peer assessment: analyzing their effect on students in a software testing course. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (pp. 231-240). IEEE. [Crossref]

[9] Eisty, N. U., Kanewala, U., & Carver, J. C. (2025). Testing research software: an in-depth survey of practices, methods, and tools. *Empirical Software Engineering, 30*(3), 81. [Crossref]

[10] Krafft, T. D., Hauer, M. P., & Zweig, K. (2024). Black-Box Testing and Auditing of Bias in ADM Systems. *Minds and Machines, 34*(2), 15. [Crossref]

[11] Adu, G. (2024). *Artificial Intelligence in Software Testing: Test scenario and case generation with an AI model (gpt-3.5-turbo) using Prompt engineering, Fine-tuning and Retrieval augmented generation techniques* (Master's thesis, Itä-Suomen yliopisto).

[12] Vaddadi, S. A., Thatikonda, R., Padthe, A., & Arnepalli, P. R. R. (2023). Shift left testing paradigm process implementation for quality of software based on fuzzy. *Soft Computing*, 1-13. [Crossref]

[13] Dadwal, A., Washizaki, H., Fukazawa, Y., Iida, T., Mizoguchi, M., & Yoshimura, K. (2018). Prioritization in Automotive Software Testing: Systematic Literature Review. *QuASoQ@ APSEC*, 52-58.

[14] Patel, J. S. (2025). AI-Driven Test Automation: Transforming Software Quality Engineering. *Journal of Computer Science and Technology Studies, 7*(2), 339-347. [Crossref]

[15] Pham, P., Nguyen, V., & Nguyen, T. (2022, October). A review of ai-augmented end-to-end test automation tools. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (pp. 1-4). [Crossref]

[16] Aghababaeyan, Z., Abdellatif, M., Briand, L., & Bagherzadeh, M. (2023). Black-box testing of deep neural networks through test case diversity. *IEEE Transactions on Software Engineering, 49*(5), 3182-3204. [Crossref]

[17] Hourani, H., Hammad, A., & Lafi, M. (2019, April). The impact of artificial intelligence on software testing. In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)* (pp. 565-570). IEEE. [Crossref]

[18] Kulkarni, N. (2020). Automated testing as part of CI/CD pipeline-shift left implementation. *North American Journal of Engineering Research, 1*(3).

[19] Anwar, N., & Kar, S. (2019). Review paper on various software testing techniques & strategies. *Global Journal of Computer Science and Technology, 19*(2), 43-49.

[20] Burgess, C. J. (2025). Software testing using an automatic generator of test data. *WIT Transactions on Information and Communication Technologies, 4*.

[21] Baqar, M., & Khanda, R. (2025, June). The Future of Software Testing: AI–Powered Test Case Generation and Validation. In *Intelligent Computing-Proceedings of the Computing Conference* (pp. 276-300). Cham: Springer Nature Switzerland. [Crossref]

[22] Andriadi, K., Soeparno, H., Gaol, F. L., & Arifin, Y. (2023, August). The impact of shift-left testing to software quality in agile methodology: A case study. In *2023 International Conference on Information Management and Technology (ICIMTech)* (pp. 259-264). IEEE. [Crossref]

[23] Gurcan, F., Dalveren, G. G. M., Cagiltay, N. E., Roman, D., & Soylu, A. (2022). Evolution of software testing strategies and trends: Semantic content analysis of software research corpus of the last 40 years. *IEEE Access, 10*, 106093-106109. [Crossref]

[24] Jalil, S., Rafi, S., LaToza, T. D., Moran, K., & Lam, W. (2023, April). Chatgpt and software testing education: Promises & perils. In *2023 IEEE international conference on software testing, verification and validation workshops (ICSTW)* (pp. 4130-4137). IEEE. [Crossref]

[25] Raksawat, C., & Charoenporn, P. (2021). Software testing system development based on ISO 29119. *Journal of Advances in Information Technology, 12*(2), 128-134.

[26] Arcuri, A., Zhang, M., Golmohammadi, A., Belhadi, A., Galeotti, J. P., Marculescu, B., & Seran, S. (2023, April). Emb: A curated corpus of web/enterprise applications and library support for software testing research. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)* (pp. 433-442). IEEE. [Crossref]

[27] AbuSalim, S. W., Ibrahim, R., & Wahab, J. A. (2021, February). Comparative analysis of software testing techniques for mobile applications. In *Journal of Physics: Conference Series* (Vol. 1793, No. 1, p. 012036). IOP Publishing. [Crossref]

[28] Kassaymeh, S., Abdullah, S., Alweshah, M., & Hammouri, A. I. (2021, October). A hybrid salp swarm algorithm with artificial neural network model for predicting the team size required for software testing phase. In *2021 International Conference on Electrical Engineering and Informatics (ICEEI)* (pp. 1-6). IEEE. [Crossref]

[29] Palani, N. (2021). *Automated Software Testing with Cypress*. Auerbach Publications. [Crossref]

[30] Bajjouk, M., Rana, M. E., Ramachandiran, C. R., & Chelliah, S. (2021). Software testing for reliability and quality improvement. *Journal of Applied Technology and Innovation, 5*(2), 40-46.

[31] Verma, A. S., Choudhary, A., & Tiwari, S. (2023). Software test case generation tools and techniques: A review. *International Journal of Mathematical, Engineering and Management Sciences, 8*(2), 293. [Crossref]

[32] Bernardo, S., Orviz, P., David, M., Gomes, J., Arce, D., Naranjo, D., ... & Pina, J. (2024). Software Quality Assurance as a Service: Encompassing the quality assessment of software and services. *Future Generation Computer Systems, 156*, 254-268. [Crossref]

[33] Atoum, I., Baklizi, M. K., Alsmadi, I., Otoom, A. A., Alhersh, T., Ababneh, J., ... & Alshahrani, S. M. (2021). Challenges of software requirements quality assurance and validation: A systematic literature review. *IEEE Access, 9*, 137613-137634. [Crossref]

[34] Pysmennyi, I., Kyslyi, R., & Kleshch, K. (2025). AI-driven tools in modern software quality assurance: an assessment of benefits, challenges, and future directions. *arXiv preprint arXiv:2506.16586*.

[35] Forgács, I., & Kovács, A. (2024). *Modern software testing techniques*. Apress. [Crossref]

[36] Pargaonkar, S. (2023). Advancements in Modern Computer Technology and Their Influence on Software Testing Practices: A Comprehensive Review. *Beyond Silicon: Advancements and Trends in Modern Computer Technology*, 221-237.

[37] Júnior, M. C., Amalfitano, D., Garcés, L., Fasolino, A. R., Andrade, S. A., & Delamaro, M. (2022). Dynamic testing techniques of non-functional requirements in mobile apps: A systematic mapping study. *ACM Computing Surveys (CSUR), 54*(10s), 1-38. [Crossref]

[38] Valle-Gómez, K. J., García-Domínguez, A., Delgado-Pérez, P., & Medina-Bulo, I. (2022). Mutation-inspired symbolic execution for software testing. *IET Software, 16*(5), 478-492. [Crossref]

[39] Boukhlif, M., Kharmoum, N., & Hanine, M. (2024, April). Llms for intelligent software testing: a comparative study. In *Proceedings of the 7th International Conference on Networking, Intelligent Systems and Security* (pp. 1-8). [Crossref]

[40] Witte, F. (2022). Strategy, Planning and Organization of Test Processes. *Wiesbaden: Springer. DOI, 10*, 978-3. [Crossref]

[41] Ali, S., & Yue, T. (2023, May). Quantum software testing: A brief introduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (pp. 332-333). IEEE. [Crossref]

[42] Bhanushali, A. (2023). Ensuring Software Quality Through Effective Quality Assurance Testing: Best Practices and Case Studies. *International Journal of Advances in Scientific Research and Engineering, 26*(1), 1-18.

[43] Ahammad, A., El Bajta, M., & Radgui, M. (2024, October). Automated Software Testing Using Machine Learning: A Systematic Mapping Study. In *2024 10th International Conference on Optimization and Applications (ICOA)* (pp. 1-6). IEEE. [Crossref]

[44] Pecorelli, F., Catolino, G., Ferrucci, F., De Lucia, A., & Palomba, F. (2022). Software testing and android applications: a large-scale empirical study. *Empirical Software Engineering, 27*(2), 31. [Crossref]

[45] Jha, P., Sahu, M., Bisoy, S. K., Pati, B., & Panigrahi, C. R. (2022, December). Application of Machine Learning in Software Testing of Healthcare Domain. In *International Conference on Advanced Computing and Intelligent Engineering* (pp. 63-73). Singapore: Springer Nature Singapore. [Crossref]

[46] Kaluarachchi, P. L., Wadasinghe, D. V., Ranaweera, E. T. M., Weerasooriya, W. M. S., De Silva, D. I., & Amarasinghe, J. V. A. A Comparative Analysis of Unit Testing and Integration Testing Based on Adding a New Feature in an E-commerce Application.

**Samia Akhtar** received her M.S. degree in Computer Science from the Virtual University of Pakistan, Lahore, Pakistan, in 2025. (Email: samiaakhtar9898@gmail.com)