TestVerse: An AI-Powered Unified Platform for Automated Software Testing using Generative Models

Abhijeet More
Professor
Department of Computer Application (MCA)
Pillai HOC College of Engineering & Technology (Autonomous)
Rasayani, India

Manjusha Jambhale
Professor
Department of Computer Application (MCA)
Pillai HOC College of Engineering & Technology
Rasayani, India

Shubham Lonkar
Department of Computer Application (MCA)
Pillai HOC College of Engineering
Rasayani, India

Soham Bhatkhande
Department of Computer Application (MCA)
Pillai HOC College Of Engineering Rasayani, India

Prem Patil
Department of Computer Application (MCA)
Pillai HOC College Of Engineering & Technology (Autonomous)
Rasayani, India

ABSTRACT
Software testing is essential for developing reliable applications. Traditional testing tools often require significant maintenance, can be unreliable, and are primarily accessible to programmers. Recently, large-scale AI models have transformed the testing landscape by enabling more efficient and integrated automation approaches [2], [4].This paper presents TestVerse, an AI-powered testing platform that integrates both black-box and white-box testing in a unified framework. The system utilizes the Google Gemini API to enhance automation capabilities. It includes features such as an AI Script Generator that converts natural language testing objectives into Selenium scripts, a Natural Language Translator for step-by-step execution, and an AI Bug Assistant that identifies errors and suggests fixes. Additionally, the platform incorporates Visual Regression Testing for UI defect detection and a White-Box Analysis module that evaluates Python code, generates pytest-based unit tests, and improves code quality. The platform aims to reduce manual effort, minimize required expertise, improve defect detection accuracy, and integrate seamlessly with CI/CD workflows.

Keywords
Automation Testing, Artificial Intelligence, Generative AI, Large Language Models (LLMs) Black-Box Testing, White-Box Testing, Selenium, Visual Regression Testing, Test Case Generation, CI/CD

1. INTRODUCTION
Software testing is a part of building any app. It plays a critical role in ensuring system reliability and makes sure users get what they expect from the app. These days with Agile and DevOps taking over the app gets. New features are added all the time so software testing matters evermore for the app [2], [5].

So teams need testing methods that actually work well and are easy to use. Sure tools like Selenium are widely used, they have some problems. Scripts stop working when the user interface changes and keeping them updated takes a lot of time. Programming knowledge is often required to use them. As a result, many teams face challenges when using these tools, with these tools [3], [5].

This is a stack setup that uses artificial intelligence to combine testing the user interface with checking the code. It does this in a way that is not normally seen [1], [4]. Users can provide testing objectives. It will create complete test scripts using Selenium. Test cases can be defined using natural language inputs in simple English and TestVerse will turn it into commands that are ready to use. The Google Gemini API is used to make this work. It gives you a lot of smart tools to use.

1. AI Test Generation (Black-Box): The system can create tests for you. You just tell it what you want to do in English and it will make a complete Selenium test script for you. You do not have to write any code.
2. AI-Assisted Debugging: When a test does not work the artificial intelligence helper looks at the error message. Tells you what is probably wrong. It even shows you how to fix the problem in your code.
3. Visual Regression Testing: The system checks the user interface to make sure it looks right. It takes a screenshot and compares it to the one that is approved. If something is wrong it finds the problem automatically.
4. AI Code Analysis (White-Box): The artificial intelligence also looks at your Python code to find problems. It suggests how to make the code and even writes complete unit tests for you using pytest. The artificial intelligence is, like an expert who reviews your code and helps you make it better. It uses intelligence to make testing easier. It is a tool that helps you test your code and user interface.

2. OBJECTIVE
This study is about creating and launching AI Powered TestVerse, which is a complete automation testing platform that uses artificial intelligence. The basic idea is to fix the issues with the testing tools and make something that is easier to use and works better.

Here's what Research is aiming for:

1. The objective is to make it easier for people to build tests even if they do not know how to program. To do this, two features are added to AI Powered TestVerse. First there is a feature that generates scripts using intelligence. This feature looks at what the user wants to do checks the webpage and then creates a script [4]. Second, users have a feature that turns English into test code that AI Powered TestVerse can run.
2. The system also focuses on detecting defects in a way. Not just looking for problems with the logic but also looking for issues with how the user interface and the code work. To do this, a feature that uses intelligence to look at the code find errors create tests and make the code better is built [2],[10].
3. Regression testing is enhanced by the system too. A system that stores, loads and replays tests making the whole process faster and more reliable [9], for AI Powered TestVerse is built.
4. Finally, the system integrates all components into a unified platform by putting the application on a real cloud server so it is ready to use for people who work with AI Powered TestVerse.
5. (Render) via a complete CI/CD pipeline based on Render and GitHub Actions.
6. To offer one web-based platform with an interactive UI for test creation, running, and producing professional PDF and Excel reports.

3. RELATED WORK

3.1 AI-Driven Test Generation Using Machine Learning

Santiago et al. proposed an advanced AI-driven approach where machine learning models are trained using data collected from human testers to automatically generate system-level test cases. The system integrates components such as classifiers, test flow models, and behavioral learning techniques to understand application states and transitions. Unlike traditional scripted automation, this approach focuses on mimicking human testing strategies, enabling more intelligent and adaptive test generation. The research highlights the importance of learning from real user interactions to improve test effectiveness. This method represents a shift from rule-based automation to data-driven testing systems, making testing more scalable and efficient.

Advantages: The system reduces manual effort in test creation and mimics human-like testing behaviour, improving test coverage and efficiency.

Limitations: The approach is still dependent on training data and cannot fully replicate human intuition in complex testing scenarios [1].

3.2 AI-Based Software Testing Techniques

Khan et al. presented a comprehensive study exploring the integration of Artificial Intelligence into software testing processes. The paper discusses various AI techniques such as machine learning, deep learning, and natural language processing, and how they can be applied to automate test case generation, defect prediction, and test optimization. It also highlights how AI can analyze large datasets to identify patterns and improve testing accuracy. The study emphasizes the growing role of AI in reducing manual effort and enhancing productivity in software development environments. Additionally, it provides insights into current trends and challenges faced in adopting AI-based testing systems.

Advantages: AI significantly enhances defect detection, reduces testing time, and improves overall software quality.

Limitations: The study is mostly theoretical and lacks real-world implementation details for large-scale systems [2].

3.3 AI-Powered Software Testing Tools

Garousi et al. conducted a systematic review of numerous AI-powered testing tools, evaluating their capabilities, functionalities, and practical applications. The study categorizes tools based on features such as automated test generation, self-healing scripts, visual testing, and failure analysis. It provides a comparative understanding of how AI is currently being used in industry-level testing environments. The research also includes empirical evaluations of selected tools, highlighting their effectiveness in real-world scenarios. This paper is valuable for understanding the current landscape of AI-driven testing solutions and their limitations.

Advantages: Provides a comprehensive overview of modern AI testing tools and their practical applications in industry.

Limitations: Most tools still require human supervision and are not fully autonomous in handling complex testing scenarios [3].

3.4 AI-Based Test Case Generation Using NLP

Agoro et al. introduced a framework that uses Natural Language Processing (NLP) and machine learning techniques to automatically generate test cases from textual requirements. The system interprets user stories or requirement documents and converts them into executable test scripts. This approach reduces dependency on manual scripting and makes test creation more accessible to non-technical users. The paper also demonstrates how NLP can bridge the gap between human language and machine-executable instructions, enabling more intuitive testing processes. This method is particularly useful in Agile environments where requirements frequently change.

Advantages: Reduces the need for programming knowledge and allows non-technical users to create test cases easily.

Limitations: The system heavily depends on input quality and may generate irrelevant or incorrect test cases in some situations [4].

3.5 Traditional Black-Box and White-Box Testing

Nidhra and Dondeti provided a detailed analysis of traditional software testing techniques, focusing on black-box and white-box testing methodologies. The paper explains how black-box testing validates system functionality without internal knowledge, while white-box testing examines the internal logic and structure of the code. It also discusses various test case design techniques such as equivalence partitioning and boundary value analysis. This research serves as a foundational study for understanding the principles of software testing and highlights the importance of combining different testing approaches for better results.

Advantages: Provides a strong theoretical foundation for understanding software testing processes.

Limitations: Lacks automation and requires significant manual effort, making it less suitable for modern development environments [5].

3.6 Automated Visual Regression Testing

Heinonen proposed an automated approach for visual regression testing that focuses on detecting unintended changes in the user interface of web applications. The method works by capturing baseline screenshots of web pages and comparing them with screenshots taken after updates or modifications. By analyzing pixel-level differences, the system identifies visual inconsistencies such as layout shifts, missing elements, color mismatches, and alignment issues. This approach complements traditional functional testing by addressing UI-related defects that are often overlooked by code-based validation. The research emphasizes the importance of maintaining UI consistency in modern web applications where frequent updates are common

Advantages: Effective in identifying UI-related defects that are often missed by functional testing.

Limitations: Sensitive to minor UI changes and requires frequent updates to baseline images [6].

3.7 AI Adoption in Software Testing

Karhu et al. conducted a study to analyze the gap between the expected benefits of Artificial Intelligence in software testing and its actual adoption in industry. The research highlights that although AI promises improved efficiency and automation, its real-world usage remains limited. Through surveys and empirical analysis, the study identifies key challenges such as lack of trust in AI systems, integration difficulties, and insufficient expertise among professionals. It also discusses organizational resistance to adopting new technologies. The paper provides valuable insights into why many companies are still hesitant to fully embrace AI-based testing solutions despite their potential advantages.

Advantages: Provides real-world insights into the adoption and effectiveness of AI in testing environments.

Limitations: Shows that AI adoption is still limited and not fully mature across industries [7].

3.8 AI in Quality Assurance Processes

Pandhare explored how Artificial Intelligence is transforming traditional Quality Assurance (QA) practices by introducing intelligent and automated solutions. The study discusses the use of advanced AI techniques such as machine learning, Generative Adversarial Networks (GANs), and reinforcement learning to improve test case design, test data generation, and defect detection. It highlights how AI can automate repetitive tasks and enhance decision-making processes in testing workflows. The research also emphasizes the potential of AI to create adaptive testing systems that evolve over time. This work demonstrates how AI can significantly improve the efficiency and effectiveness of QA processes in modern software development.

Advantages: Improves efficiency and enables smarter testing strategies.

Limitations: Implementation complexity and dependency on advanced AI models limit practical usage [8]

3.9 AI-Based Regression Testing Optimization

Khaleel et al. proposed an AI-based approach to optimize regression testing by prioritizing and selecting the most relevant test cases. The system uses machine learning algorithms to analyze historical test data and identify which test cases are most critical for detecting defects. This helps reduce the time and computational resources required for regression testing, especially in large-scale applications. The study demonstrates how intelligent test selection can maintain software quality while improving efficiency. The approach is particularly useful in continuous integration environments where frequent updates require repeated testing.

Advantages: Reduces testing cost and improves execution efficiency.

Limitations: Requires historical data and may not perform well for new applications [9].

3.10 Role of AI in Software Testing (Future Perspective)

Hayat et al. explored the future potential of Artificial Intelligence in transforming software testing processes. The paper discusses how AI techniques such as machine learning and deep learning can be integrated into various phases of the testing lifecycle, including test generation, execution, and maintenance. It highlights the concept of autonomous testing systems where minimal human intervention is required. The study also examines challenges such as trust in AI decisions, interpretability of models, and integration with existing systems. Overall, the research provides a forward-looking perspective on how AI will shape the future of software testing.

Advantages: Highlights future trends and the growing importance of AI in testing.

Limitations: Challenges such as integration complexity and trust in AI systems remain unresolved [10].

3.11 Evaluating LLM-Based Test Generation Under Software Evolution

Haroon et al. perform a large-scale empirical study of LLM-based unit test generation in the face of code changes. They use a mutation-driven framework on 22,374 program variants and eight state-of-the-art LLMs to evaluate test suites under semantic-altering and semantic-preserving changes. Their analysis shows that LLMs achieve high coverage on original code (∼79% line coverage) but degrade significantly under evolution (e.g., pass rates fall to 66% after semantic changes). Most failing tests still pass on the original program, indicating LLMs often rely on surface patterns rather than true semantic understanding. The study highlights that LLM-generated tests struggle to maintain regression awareness as software evolves.

Advantages: Extensive dataset (22k+ variants) and multiple LLMs evaluated. Distinguishes between semantic vs. non-semantic code changes. Quantifies LLM weaknesses (coverage drop under evolution).

Limitations: ArXiv preprint (not yet peer-reviewed). Focus is on small programs/unit tests, not full systems. No proposed solutions (descriptive analysis only).[11]

3.12 A System for Automated Unit Test Generation Using Large Language Models and Assessment of Generated Test Suites

Lops et al. introduce AgoneTest, a fully automated framework that uses LLMs to generate and evaluate unit test suites for Java classes. To scale beyond method-level tests, they extend the Methods2Test benchmark into a new Classes2Test dataset mapping Java classes to test classes. AgoneTest automates the entire cycle: it feeds class code to LLMs (via prompt templates), generates class-level JUnit tests, and uses tools like

JaCoCo and PIT to compute coverage and mutation metrics. Evaluated on real GitHub projects, AgoneTest demonstrates that LLMs can produce plausible class-level tests when driven by a structured pipeline. This work fills a gap by automating test generation and assessment at realistic scale.

Advantages: End-to-end automated pipeline for LLM-based test generation. New Classes2Test dataset linking classes to human-written tests. Integrates coverage and mutation tools (JaCoCo, PIT) for objective evaluation.

Limitations: Limited to Java (no multi-language evaluation). Depends on specific tooling (PIT, etc.) which may not apply everywhere. Dataset in evaluation (94 classes) is modest size. Does not explore interactive or self-healing test adjustments. [12]

3.13 Enhancing LLM-Based Test Generation by Eliminating Covered Code

Xu et al. propose a novel two-step framework to improve coverage for LLM-generated unit tests on complex methods. First, they extract context (dependencies, helper functions) to create a focused code slice. Second, they perform iterative test generation: an LLM (e.g. GPT-4o) generates tests for the slice, then covered code segments are removed before the next iteration. This "divide-and-conquer" mitigates token limits and keeps the LLM focused on uncovering new behavior. In evaluations on open-source Python projects, this method significantly outperforms baseline LLM and search-based testers, achieving much higher line coverage on complex functions. The results show that systematically eliminating already-tested code guides the LLM to explore remaining paths.

Advantages: Boosts test coverage on complex, large methods. Addresses LLM token/complexity limits via iterative code slicing. Demonstrated empirical superiority over state-of-the-art baselines. General approach (theoretically applicable to any LLM).

Limitations: Multi-round LLM calls incur higher cost and latency.

Implemented for Python; adaptation needed for other languages. Requires careful static analysis to slice code (may be brittle).Depends on quality of the underlying LLM (GPT-4o used in evaluation). [13]

3.14 Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts

Yaraghi et al. tackle regression test case prioritization in real CI environments. They first define a comprehensive data model capturing CI data (code changes, test history, coverage, build outcomes, etc.) and derive an extensive feature set from it. They collect data from 25 open-source projects (chosen for sufficient regression time and failures) and build ML models to order test cases to detect faults early. Their study answers questions about feature importance, model decay over time, and trade-offs between data collection costs and prioritization gains. Key findings include: ML-based prioritization can significantly reduce time to fault detection, but model effectiveness slowly degrades as the codebase evolves. This work provides deep insights into practical ML-driven prioritization.

Advantages: Broad CI feature set and data model, covering code, test, and build aspects. Large empirical dataset (25 diverse projects with real failure history). Rigorous evaluation of ML models and feature impacts. Published in IEEE TSE (highly reputable venue).

Limitations: Designed for CI pipelines; may be less applicable outside CI (e.g., ad-hoc testing). Data collection and feature computation can be costly in practice. Focuses on prioritization (not test generation or fixing). Results might not generalize to proprietary or very different codebases.[14]

4. RESEARCH GAPS

4.1 The Test Case Generation & Accessibility Gap

Creating test cases for software testing takes a lot of time. You cannot skip this step. Most automation tools do not make it easier to create test cases. Engineers have to look at the app and find the IDs or classes. Then they have to write scripts. These scripts often do not work when someone changes the interface or the logic. This is a process. It is hard for team members who do not know how to code to help with test cases. If you are not technical it is very hard to help [5].

Gap: There is a problem with some automation tools. Some AI tools try to read and understand the requirements for software testing. Most of these tools cannot handle real apps. There is a gap in the tools that're available. Few tools can look at web pages. Turn simple instructions into useful scripts [3][4], for test cases. Test cases are a part of software testing. There are not tools that let people who do not know how to code create test cases. These tools should let people use commands that are easy to understand. Software testing needs test cases. Test cases need to be easy to create.

How AI Powered TestVerse Fills This Gap: Platform directly addresses this by providing two Generative AI features:

AI Script Generator: You just drop in a URL and tell the AI what you want to do like "Test the login process." The AI checks out the live HTML on that page and spits out a full Selenium script, step by step, ready to run.

1. **Natural Language Translator**: You write out what you want, line by line maybe something like "click the login button." The AI takes that and turns it into the exact command the tool needs, no guesswork.

4.2 The Maintenance & Debugging Gap

Automation scripts can be really unreliable. Sometimes even a small change in the user interface or some unexpected content can cause problems. A tiny delay can also make the tests unsteady and costly to maintain [3]. When a standard test fails the error message is often unclear like NoSuchElementException. So someone has to investigate and find out what actually caused the issue, with the automation scripts. The person has to look into the automation scripts and figure out what went wrong with the automation scripts.

Gap: Most systems just leave you to figure things out when something goes wrong. Sometimes a self-healing system tries to fix problems by itself. It's really hard to understand why your test failed. Usually developers have to dig through a lot of logs to try and guess what went wrong. They have to look at the logs to find the problem [3], [7]. Self-healing systems try to fix things Developers do not have to guess what went wrong, with self-healing systems. They can just see the fix happen. Most systems do not work that way. They just. Leave you to fix them.

That can be really frustrating. You have to find another locator yourself. That is work. Self-healing systems make it easier. They save developers a lot of time. They fix things fast. That helps a lot.

That's where TestVerse comes in. Built a tool that actually helps. When a test step fails platform collects the error message, the failed command and the webpage address. Now fixing problems and keeping things running smoothly is much easier, with TestVerse. The AI looks at everything. Tells you what probably went wrong. It also gives you a fix to try. You get answers quickly instead of wasting time finding bugs. Now fixing problems and keeping things running smoothly is much easier, with TestVerse.

4.3 The Lack of Integrated, Full-Spectrum Analysis

Most testing tools are pretty specialized. They usually do one thing [3][6]. For example some tools do white-box testing. They run static code analysis with linters. Other tools do black-box testing. They use Selenium, for UI testing. Visual testing is different. It spots layout or colour issues. Most visual testing tools are services. They are often expensive too.

Gap: A lot of testing tools do not work together. This makes workflow messy. Developers jump between tools to find bugs UI problems and code issues. It gets really old fast. There is a need of one platform to handle UI and code testing. That way everything will run smoothly. Nobody will waste time switching between tools. Can focus on fixing bugs not searching for them. One platform will make lives easier.

How TestVerse helps with this problem is that it has a testing system:

1. It can do Black-Box Testing using a tool that is based on Selenium.
2. It also does Visual Regression Testing, which means you can see how things looked before and spot any changes in the user interface away.
3. TestVerse has a White-Box Analysis system. The AI in TestVerse looks at the code for TestVerse to find mistakes makes it better for faster performance and even makes full unit tests, for TestVerse.

5. SYSTEM ARCHITECTURE AND TECHNOLOGIES USED

The AI Powered Automation Testing Platform is made using an idea [2]. It combines the parts that users see and the parts that work behind the scenes. The background part does all the work, like handling data and running tests. The user sees the part, which is made to be easy to use and understand.

5.1 Technologies

1. Python

Using Python to make the backend work because it has a lot of tools that make it easy to automate things and do machine learning. Python is also very flexible and easy to understand. Python is used due to its flexibility and strong library support to create the backend logic. Python is considered an appropriate choice because it has strong library support. Python is used to implement the backend due to its flexibility and strong library support.

2. Flask:

This web framework is highly effective for making a web server. It helps with routing and managing requests and responses, in a way. The web framework makes it easy to handle all of this.

3. SQLite:

A file-based relational database that stores user accounts, test case histories, setups, and results data. It makes it easy to use and move small to medium-sized applications.

4. Flask-SQLAlchemy:

SQLAlchemy is an ORM tool that makes it easy for databases to talk to each other by letting Python objects and the SQLite database talk to each other easily.

5. Selenium used for testing:

The main automation engine that does black-box testing tasks like clicking, typing, and checking how people use online apps [3].

6. Pillow library:

A Python image library used for visual regression testing to compare screenshots and find UI differences between versions.

7. Library fpdf2 & pdfkit :

These libraries make it easier to keep track of and document things by making downloadable PDF reports that summarize black-box and white-box testing.

8. The openxyl library (python):

It is used for Excel sheet. It also gives you the option to look at test results and export them which's really helpful for business reporting. The prepared test scripts can be exported in to Excel sheet report using this library.

9. Gunicorn library:

A production web server is used for running the Flask application efficiently in cloud platforms like Render.The server helps to make sure the application runs smoothly. The Flask application is run securely and efficiently. Cloud platforms such, as Render are supported.

6. METHODOLOGY

The AI Powered TestVerse platform follows a structured workflow that integrates artificial intelligence with automated testing techniques. The methodology is designed to simplify the testing process while improving efficiency and accuracy.

The process begins with user input, where testing requirements are provided in natural language or as source code. These inputs are processed using the AI model, which interprets the intent and generates appropriate outputs such as Selenium test scripts, debugging suggestions, or unit test cases.

In the case of black-box testing, the AI Script Generator converts natural language instructions into executable Selenium scripts. These scripts are then executed using the automation engine to validate application functionality.

For white-box testing, the system analyzes the provided source code to identify potential issues and generate unit test cases using frameworks such as pytest. This enables validation of internal logic and improves code reliability.

The Visual Regression Testing module captures and compares UI screenshots to detect visual inconsistencies. This helps identify layout changes, missing elements, and other UI-related defects that are not detected by functional testing.

Finally, the system generates detailed reports in PDF and Excel formats, providing insights into test execution, detected defects, and overall performance. This structured workflow ensures efficient and intelligent automation testing.

6.1 System Design Overview

The system is built on a modular client-server systemThe backend, which is written in Python and Flask, does all the thinking and reasoning, while the frontend uses HTML, CSS, and JavaScript to make the interface interactive.

The system architecture involves three main components:

1. AI Engine (Google Gemini): Handles all the intelligent operations, such as AI-based test script generation, language translation, bug analysis, and white-box code checking.
2. Automation Core: which uses Selenium runs tests, on web browsers. It does things like click buttons and type text. For Automation the use of library “pillow” can be used.
3. Reporting and Visualization Layer: The frontend shows test results in a different way. It uses tools like fpdf2 and pdfkit, for PDFs and openpyxl for Excel files.

6.2 System Workflow

Development and operational workflow of the platform encompass the following key phases:

1. **Test Case Input:** Three flexible ways of test creation are offered by the system:
2. **Manual Commands:** Users can provide input commands to do like click on something with a name for example click on the thing that says "id=login".
3. **Figure 3: User** registration **interface for new tester**
4. **Natural Language Translation:** Users can provide instructions in natural language like "click the login button" and then the "Translate" thing will turn it into a command that the computer can understand.
5. **AI-Based Test Case Generation:** You can give it a website address. Tell it what you want to do like "Test the login process".

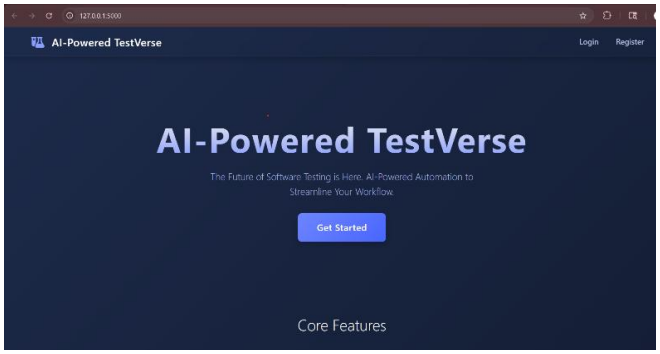

Figure 1 Home Page of AI Powered TestVerse showing the main navigation interface.

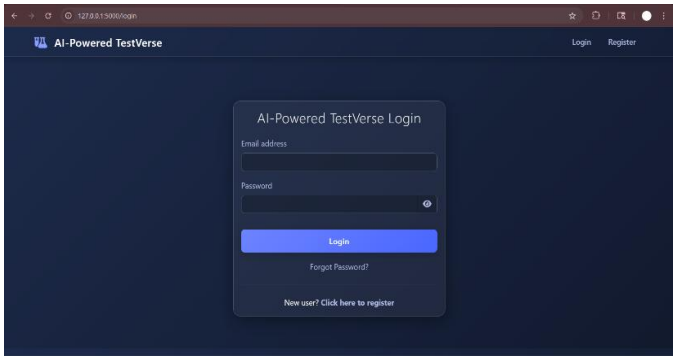

Figure 2 Secure Login Using Flask Authentication

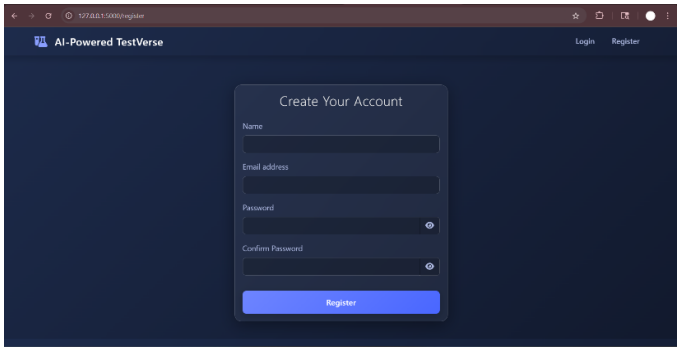

Figure 3: User registration interface for new tester

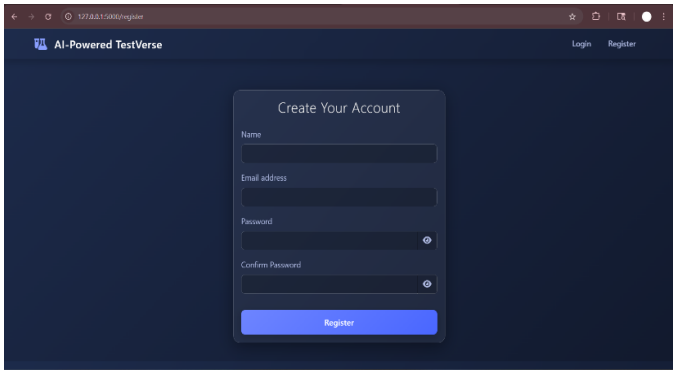

Figure 4: Password-recovery module integrated in the authentication system

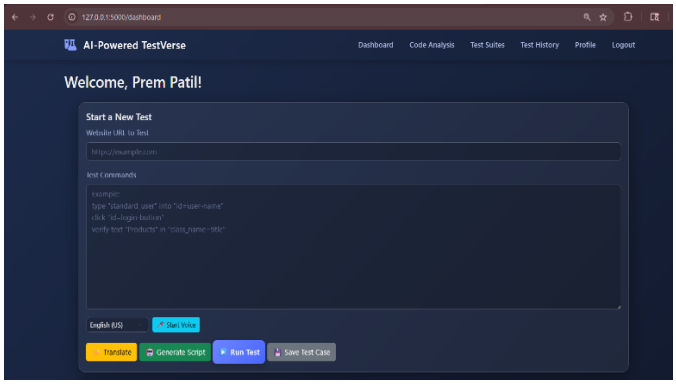

Figure 5 Dashboard for user test creation and reports

The tool also lets us make things happen on the website like showing results doing something when you click a button and controlling messages that pop up on the screen.

The tool also lets us make things happen on the website like showing results doing something when you click a button and controlling messages that pop up on the screen.

6.3 Evaluation Metric

1. **Test Creating Efficiency:** The time required to generate a complete test script with multiple steps was evaluated. A comparison was conducted between manual test script creation and the use of the AI Script Generator. The results indicate that the AI Script Generator reduces the time required by over 90 percent [4].
2. **AI Generation Accuracy:** The accuracy of AI-generated scripts was evaluated by measuring how often the generated scripts executed successfully without requiring modifications.
3. **AI Bug Assistant Utility:** Failed test cases were analyzed to determine whether the AI Bug Assistant correctly identified the root cause of errors and provided appropriate solutions.
4. **Defect Detection Rate:** The system's ability to detect different types of defects was evaluated, including logic errors, UI-related issues, and code quality problems.
5. **Accessibility for Non-Technical Users:** The usability of the system for non-technical users was evaluated by analyzing how effectively users without programming knowledge could create and execute test cases using Natural Language Translation and AI Script Generator features.

7. RESULTS AND DISCUSSION

To evaluate the performance of the proposed AI-Powered TestVerse platform, a series of experiments were conducted on web-based applications. The evaluation focused on key performance metrics such as test creation time, accuracy of generated scripts, defect detection capability, and usability for non-technical users. The experimental results were compared with traditional manual testing approaches to analyze improvements in efficiency and effectiveness.

Table 1: Comparison of Traditional Testing and AI-Powered TestVerse

Metric	Traditional Testing	AI-Powered TestVerse	Improvement
Test Creation Time	10–15 minutes	< 1 minute	~90% faster
Script Accuracy	Medium	High	Improved
Defect Detection	Functional only	Functional + UI + Code	Enhanced
Debugging Effort	High	Reduced with AI Assistant	Significant reduction
Accessibility	Requires programming	No programming required	Highly accessible
Report Generation	Manual	Automated (PDF & Excel)	Fully automated

Table 2: Performance Evaluation of AI Modules

Module	Performance	Observation
AI Script Generator	High Accuracy	Generates executable scripts quickly
Natural Language Translator	High	Converts user input effectively
AI Bug Assistant	Moderate–High	Identifies root causes and suggests fixes
Visual Regression Testing	High	Detects UI inconsistencies
White-Box Analysis	High	Generates unit tests and improves code quality

Table 3: Usability Evaluation for Non-Technical Users

Criteria	Result
Ease of Test Creation	High
Understanding of Outputs	High
Success Rate (Test Execution)	85–90%
Learning Curve	Low

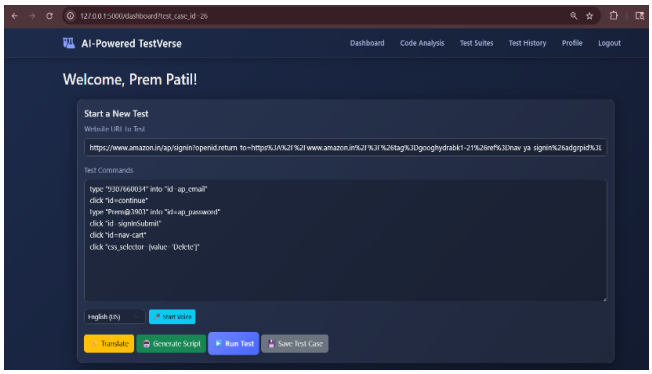

Figure 6: complete two-step wise execution process ouput

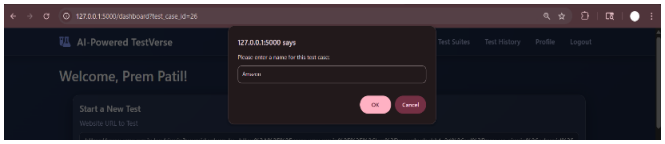

Figure 7: Interface for saving completed test cases to the SQLite database.

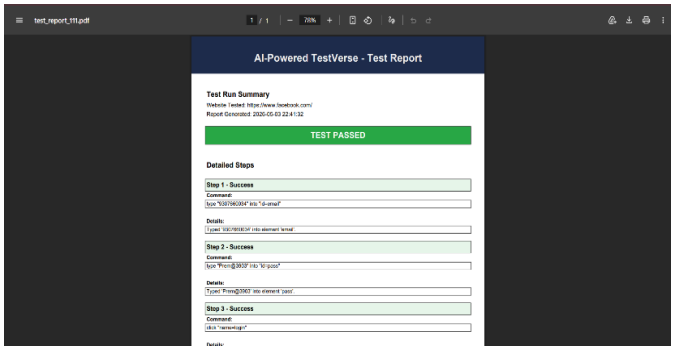

Figure 08: Auto-generated PDF report summarizing black-box and white-box tests.

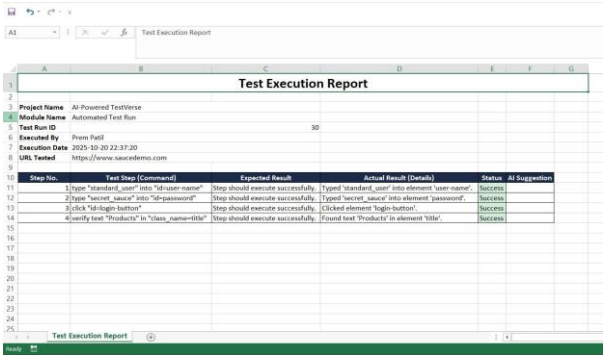

Figure 09: Excel report generated using openpyxl for detailed analysis.

7.1 Performance Evaluation

The results were analyzed to evaluate system performance. The study evaluated how test scripts are created and compared it to manual testing approaches. This was done to find out how effectively the AI-driven testing system really performs. The objective was to determine whether the AI-driven testing approach is better than developing test scripts by hand. The study evaluated the process of test script generation and compared it with manual testing approaches. The objective was to determine the effectiveness of the AI-driven testing system. The evaluation considered factors such as test creation time, defect detection capability, and maintenance effort.

Metric	Traditional Manual Testing	AI Powered TestVerse (AI-Powered)	Improvement
Test Case Generation Time	10–15 mins (Manual Inspection & Coding)	< 1 minute (Using AI Script Generator)	**~90% faster**
Script Maintenance Effort	High (Manually debugging error logs)	Low (AI Bug Assistant suggests fixes)	Drastically Reduced Debugging Time
Defect Detection Scope	Functional bugs only.	Functional, **Visual**, and **White-Box** (code quality) bugs.	Greatly Expanded Scope
Accessibility	Requires skilled automation engineers (coders).	Accessible to non-technical users via Natural Language commands.	High
Report Generation	Manual / Semi-automated.	Fully automated **PDF** and **Excel** reports.	100% Automated

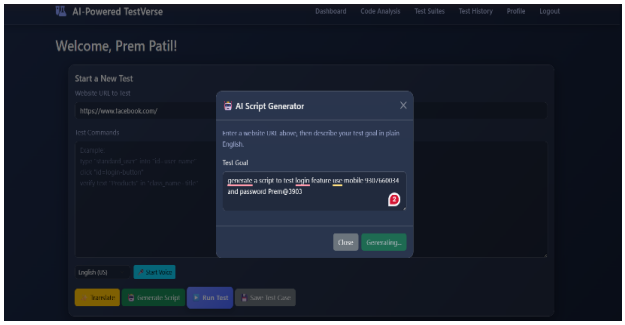

Figure 10: AI Script Generator input interface for goal entry.

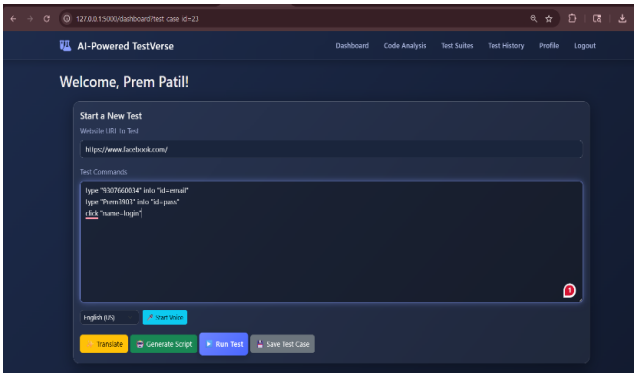

Figure 11: AI Script Generator output showing auto-generated Selenium code.

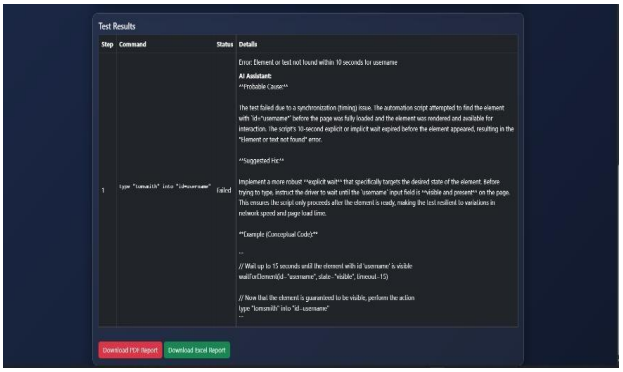

Figure 12: AI Bug Assistant displaying probable cause and suggested fix for failed tests

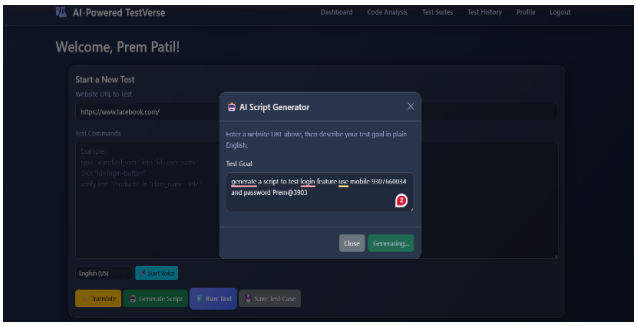

Figure 13: Sample Prompt

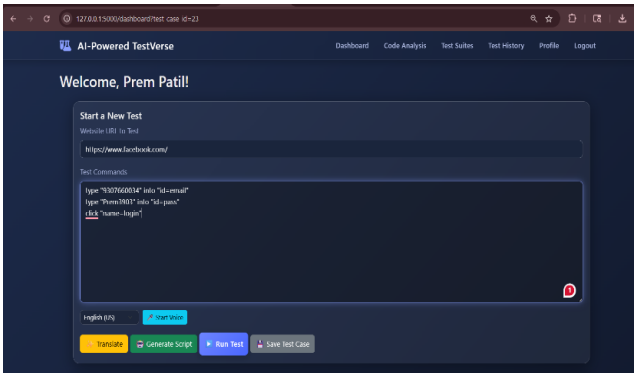

Figure 14: AI Converting the prompt to selenium supported script

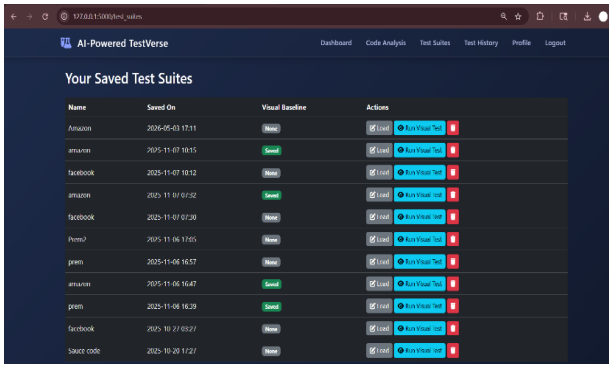

Figure 15: Saved test suites available for re-execution

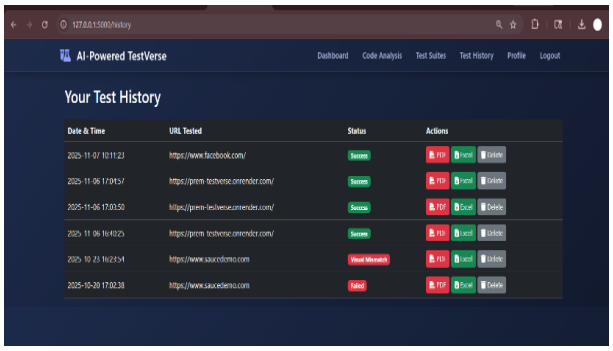

Figure 16: Test log for all the history Testscripts

1. **Generator** produced a correct, ready-to-run script in under a minute.

2. **Accuracy & Accessibility:** The artificial intelligence tool is really good at taking English and turning it into scripts [4] that the computer can understand. It does this with the Translate and Generate Script features. The artificial intelligence tool is very accurate. It can find the selectors like data-test. It can also build scripts. This makes it easier, for people who do not know how to program to automate things. The artificial intelligence tool lowers the barrier for automation. Non-programmers can use the intelligence tool to participate in automation.

3. **Comprehensive Test Coverage:** The platform successfully expanded testing beyond simple functional checks. The **Visual Regression Testing** module was effective at identifying UI defects [6] (e.g., "Visual Mismatch"), and the **White-Box Analysis** module successfully analyzed Python code to find bugs and refactor it for better performance.

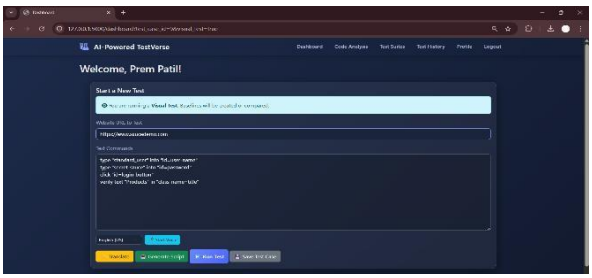

Figure 17 a: Regression testing of interfaces.

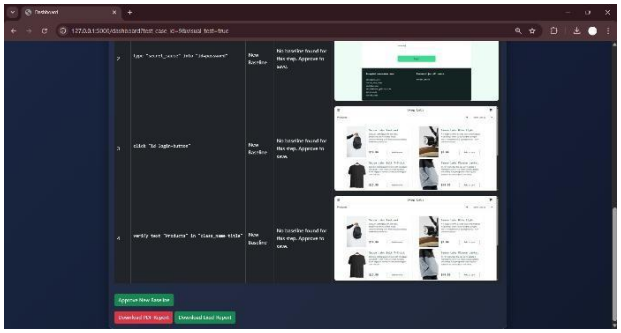

Figure 17 b: The Output for standard visual mechanics.

7.2 Result Analysis

The results illustrate that the AI-powered platform resoundingly beats traditional automation techniques in several key areas:

1. **Efficiency:** The system dramatically shortens test case generation time. The results indicate that it was clear that making a test script by hand took more than 10 minutes while the AI Script took a lot less time to do the same thing. The AI-generated script demonstrated significantly higher efficiency, at creating the test script than manual test script creation.

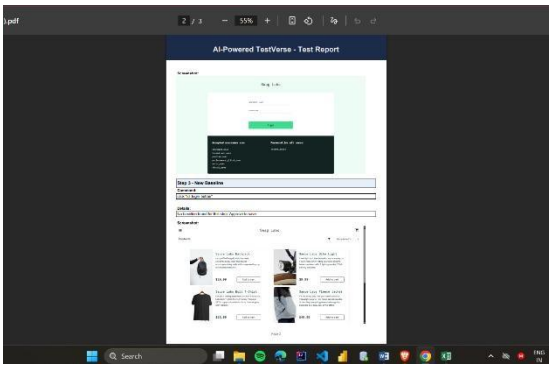

Figure 18: Download report in PDF Format

2. **Comprehensive Reporting:** When you use the automated system to make PDF and Excel reports you get the documents away. These papers list everything that happened during the exam. You can see the actions that were taken and the tests current state. This keeps records of every test run of the exam, which's really helpful for the PDF and Excel reports. The automated system makes PDF and Excel reports, for every test run of the exam. That is how you get the documentation you need from the PDF and Excel reports.

3. **Integration with CI/CD Pipelines:** The tool was set up to work with GitHub Actions and Render. This made it possible to put the tool on a live Render environment. This shows that the tool can work well with a continuous integration workflow. The tool integration, with CI/CD Pipelines was a success. The tool works with GitHub Actions and Render to make this happen.

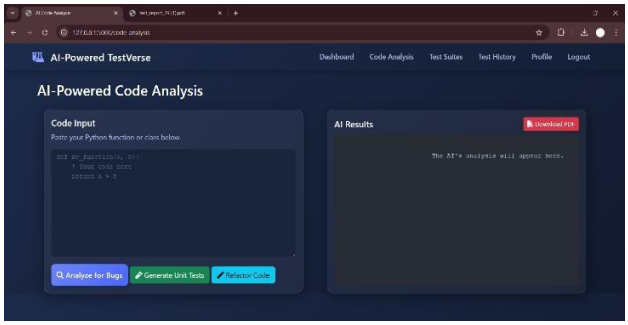

Figure 19a: White-Box Testing module for code input.

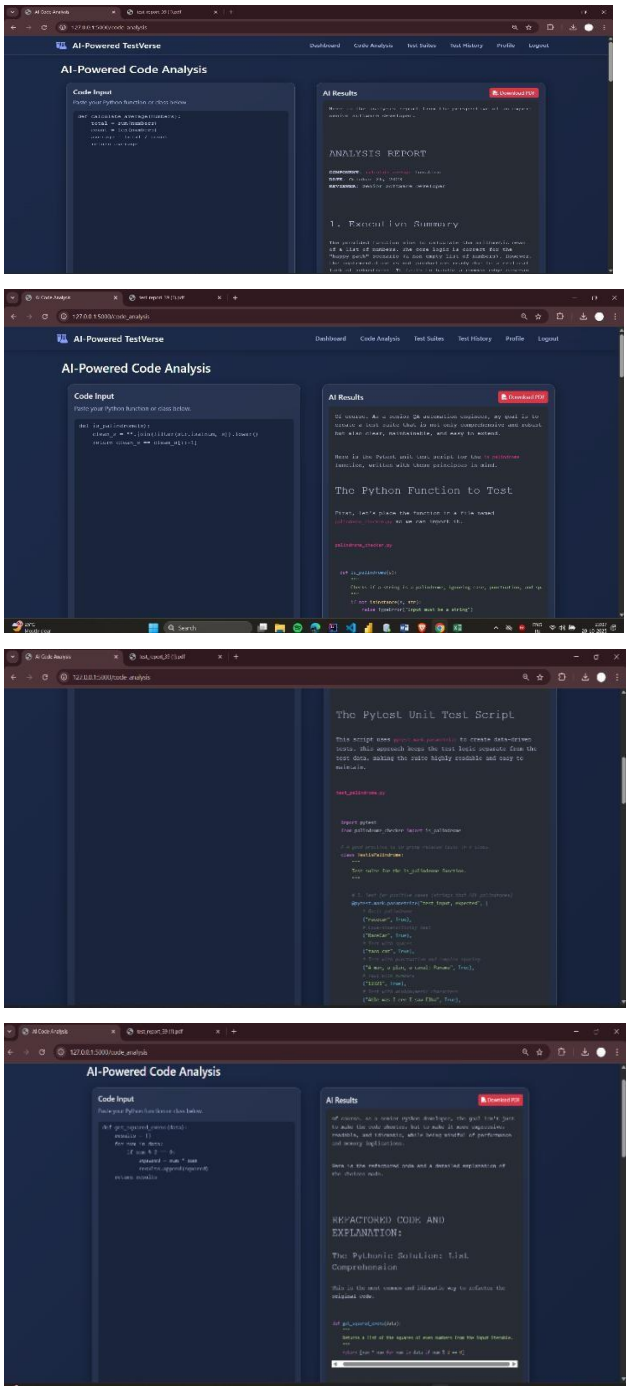

Figure 19b: Refactored code output.

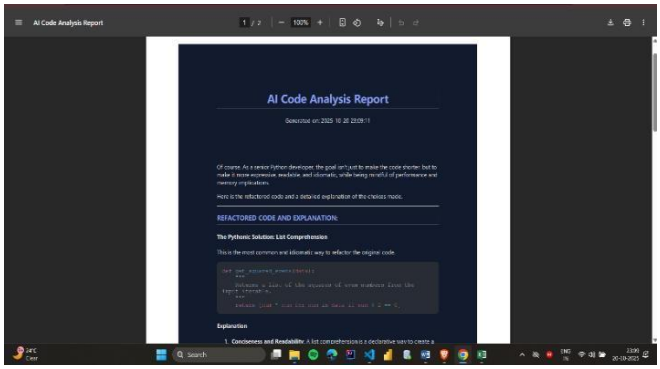

Figure 20: PDF report generated from AI-based White-Box Analysis.

7.3 Discussion

The results show how Generative AI can change software testing for the better. By adding the Google **Gemini API** to the testing process the platform combines old-style automation with help. This reduces the need for manual test creation and fix tests.This reduces manual effort and minimizes errors. [2],[8]. The role of a tester evolves from writing scripts to being a "test strategist" who uses AI. The AI still needs clear goals from the user to create good test cases. The AIs ability to generate test cases also depends on how it is trained. The system could get better if it becomes an "AI Exploratory Agent". This agent would work on its own to find bugs. It would have a goal to search for problems, in the code.

8. CONCLUSION AND FUTURE SCOPE

8.1 Conclusion

The AI-Powered Automation Testing Platform, The AI-Powered TestVerse platform integrates generative AI and traditional automation models. This makes testing more intelligent, efficient and user-friendly [2], [10]. The platform uses technologies, like Python, Flask, Selenium and the Google Gemini API. These tools help test case execution. They also make it easy to create, debug and maintain test cases. AI Powered TestVerse is changing the way users do testing with AI. It makes testing faster and smarter. In addition to this the platform can create PDF and Excel reports. It also works well with a CI/CD pipeline. Further-more it supports features like voice commands and AI-driven code analysis. These features together make the system a complete and intelligent testing solution. This research contributes to the growing field of AI-enhanced software quality assurance. The results demonstrate the benefits of intelligent automation in modern DevOps environments. The platforms capabilities make it a valuable tool for software testing.

9. FUTURE SCOPE

This study demonstrates promising results but there are still a lot of ways to make the platform even smarter and better. The platform can be improved in ways to make it more useful.

1 **Implementation of an AI Exploratory Testing Agent:** The next step is to go from testing that is helped by Artificial Intelligence to testing that is done on its own. A future version might use an Artificial Intelligence agent that can look at the website by itself make decisions and find problems without needing a script that was written ahead of time when it is given a general goal such as testing the checkout process. The Artificial Intelligence Exploratory Testing Agent will be able to do this kind of testing.

2 **Integration of Advanced LLM models:** Future improvements can enhance the system at understanding what people are saying. To do this Future work may include the language models as they come out. This will help us get better at understanding what people mean. Will be able to explain mistakes in a way that's easy to understand. Users should keep using the language models. These include the LLMs. They will make system more effective, at understanding language. It will also give explanations of bugs.

3 **AI-Driven Security Testing:** Future work should include the addition of a feature that lets the AI act like a security expert. This new feature will help the AI find problems in code, such as SQL injection and Cross-Site Scripting also known as XSS. The AI will also look for ways that data is handled. The AI will act as a security expert to check code for problems like SQL injection. The AI will also check code for Cross-Site Scripting, which is also known as XSS. In addition, to these the AI will look for data handling.

10. REFERENCES

[1] D. Santiago, T. M. King, and P. J. Clarke, "AI-Driven Test Generation: Machines Learning from Human Testers," 2017.

[2] S. A. Khan et al., "AI-Based Software Testing," 2024.

[3] V. Garousi et al., "AI-powered software testing tools: A systematic review," 2023.

[4] H. Agoro and A. Matthew, "AI-Powered Test Case Generation and Execution," 2023.

[5] S. Nidhra and J. Dondeti, "Black box and white box testing techniques – A literature review," 2012.

[6] J. Heinonen, "Design and Implementation of Automated Visual Regression Testing," 2020.

[7] K. Karhu et al., "Expectations vs Reality: AI Adoption in Software Testing," 2025.

[8] H. V. Pandhare, "From Test Case Design to Test Data Generation: AI Redefining QA," 2024.

[9] S. I. Khaleel and R. Anan, "Optimal Test Cases for Regression Testing using AI Techniques," 2023.

[10] M. A. Hayat et al., "The Evolving Role of AI in Software Testing: Prospects and Challenges," 2024.

[11] S. Haroon, M. T. Khan, and M. A. Gulzar, "Evaluating LLM-Based Test Generation Under Software Evolution," arXiv preprint, 2026.

[12] A. Lops et al., "A System for Automated Unit Test Generation Using Large Language Models and Assessment of Generated Test Suites," IEEE ICST Workshops (ICSTW), 2025.

[13] W. Xu, M. Liu, and F. Kong, "Enhancing LLM-Based Test Generation by Eliminating Covered Code," arXiv preprint, 2026.

[14] A. S. Yaraghi et al., "Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts," IEEE Trans. Software Eng., vol. 49, no. 4, pp. 1615–1639, 2023.