International Journal of Information Technology & Computer Engineering

ISSN 2347–3657

Volume 13, Issue 4, 2025

An Intelligent Multi-Stage Framework For Web Application Testing: Risk-Based Compatibility Assessment And Reinforcement Learning-Driven Performance Optimization

[1]D. Mythily, Dr. N. Kamaraj[2]

[1]Research Scholar, Department of Computer Science, Sri Ramakrishna Mission Vidyalaya College of Arts and Science, Coimbatore, Tamilnadu, India.

[2]Head and Assistant Professor, Department of Information Technology, Sri Ramakrishna Mission Vidyalaya College of Arts and Science, Coimbatore, Tamilnadu, India.

ABSTRACT

In modern web application development, ensuring cross-platform compatibility and optimal performance has become increasingly challenging due to the diversity of devices, browsers, and user loads. This paper proposes an intelligent two-level testing framework integrating machine learning and reinforcement learning to address these challenges. In the first level, a compatibility testing model utilizes historical defect data and platform-specific features to train a Random Forest classifier that predicts high-risk platform combinations, effectively reducing the total number of test cases by 40%. In the second level, a performance testing model employs Q-learning to dynamically explore and identify the system's safe operational boundaries under varying load conditions. Experimental results demonstrate the effectiveness of the suggested strategy in enhancing flaw detection, optimising test resources, and self-learning system limitationswithout manual intervention. This hybrid intelligent framework significantly enhances the reliability, scalability, and cost-effectiveness of web application testing.

Keywords: *Web application, machine learning, reinforcement learning, compatibility testing, performance testing and Random Forest classifier.*

1. INTRODUCTION

Web applications have become the backbone of modern businesses, offering services across diverse platforms, devices, and networks. Unlike traditional software systems, web applications operate in highly heterogeneous environments with multiple browsers (Chrome, Firefox, Edge, Safari), devices (PCs, tablets, smartphones), operating systems (Windows, macOS, Android, iOS), and network conditions (WiFi, 4G, 5G, varying latencies). As a result, ensuring that a web application performs consistently across all these environments is a significant challenge for software testers and quality assurance engineers.Two critical testing levels are essential to ensure the quality of web applications:

- **Compatibility Testing:** Verifies that the web application functions correctly across multiple combinations of browsers, operating systems, devices, and screen resolutions.
- **Performance Testing:** Validates that the application responds appropriately under different load conditions, including peak usage, stress scenarios, and long-duration endurance tests.

While much focus in software testing research has been placed on functional correctness and security, compatibility and performance testing are often treated as secondary levels due to their complexity and high resource requirements. However, compatibility and performance issues directly affect the end-user experience, leading to user dissatisfaction, financial losses, and reputational damage.

1.1 Importance of Compatibility Testing

Compatibility testing ensures that the application behaves consistently across diverse client platforms. The growing diversity in user platforms—especially mobile device fragmentation—has increased the importance of compatibility testing. Issues such as improper rendering, broken layouts, browser-specific JavaScript errors, and CSS incompatibility often arise in real-world deployments. Furthermore, third-party plugin conflicts, localization issues, and hardware limitations can introduce subtle defects not easily discovered during functional testing.

1.2 Importance of Performance Testing

Performance issues such as high response times, server crashes, resource exhaustion, and unresponsiveness can critically damage user trust. Modern web applications often serve thousands or millions of concurrent users globally, making load balancing, caching, database optimization, and network optimization crucial factors in application performance.

Performance testing includes:

- Load Testing
- Stress Testing
- Spike Testing
- Endurance (Soak) Testing
- Scalability Testing

Each of these evaluates different aspects of system robustness and scalability.

1.3 Motivation

The traditional approach to compatibility and performance testing is exhaustive, costly, and time-consuming. The number of possible browser-device-OS combinations grows exponentially with each new version released. Furthermore, simulating real-world load profiles in performance testing is challenging without accurate user behavior data.

Current challenges include:

- Large number of test combinations for compatibility testing.
- Lack of real-world prioritization of critical test scenarios.
- Inaccurate or artificial performance load profiles that do not reflect true user behavior.
- Lack of intelligent test selection mechanisms to reduce testing effort.

1.4 Research Objectives

This research's principal goal is to createan intelligent and automated framework for compatibility and performance testing of web applications. Specifically, the research aims to design a machine learning-based compatibility testing model that can predict high-risk platform combinations using historical defect data and various platform-specific features. Additionally, it seeks to build a reinforcement learning-based performance testing model that can dynamically identify the system's safe operational limits under varying workloads without manual trial-and-error processes. By achieving these objectives, the research intends to optimize testing efforts, improve defect detection efficiency, reduce manual intervention, and enhance Web application testing's overall scalability and dependability.

1.5 Research Contributions

The research involved adds a number of significant insights to the subject ofsoftware quality assurance. First, it introduces a risk-based compatibility testing approach utilizing Random Forest classification, which effectively prioritizes platform combinations and reduces unnecessary test executions. Second, it proposes a novel application of Q-learning for performance testing, enabling the system to autonomously learn and identify optimal load thresholds. Third, the integration of machine learning with automated testing tools offers a highly adaptive, efficient, and cost-effective solution for complex web application testing scenarios. Finally, The experimental findings confirm the usefulness and efficiency of the suggested framework, offering a strong basis for next developments inintelligent software testing methodologies.

2. LITERATURE SURVEY

2.1 Adaptive Test Prioritization Using User Behavior Analytics

Chen et al. (2020) proposed an adaptive test case prioritization framework that uses real-time user behavior analytics (UBA) collected from production systems. Their model assigns higher priority to test cases that cover features most frequently used by end users. The approach dynamically adjusts test priorities based on evolving user interactions, helping compatibility and performance tests focus on high-impact areas. This method efficiently reduces unnecessary testing of rarely used features while ensuring critical functionality is thoroughly tested.

2.2 ML-Based Browser Compatibility Testing

Gupta et al. (2021) introduced a machine learning method for compatibility with browserstesting. They trained a Random Forest model on historical defect data, DOM structures, and browser engine differences to predict likely compatibility issues across multiple browsers. The model identifies high-risk page elements prone to cross-browser failures, enabling selective testing and faster defect detection. This data-driven method minimizes redundant full-matrix compatibility tests.

2.3 Reinforcement Learning (RL)

Huang et al. (2021) proposed an AI-driven performance testing framework using Reinforcement Learning (RL). Their framework automatically learns optimal workload patterns for performance testing by interacting with the system under test. The RL agent continuously adjusts the load to identify performance bottlenecks under realistic traffic scenarios. This adaptive mechanism overcomes the limitations of static load profiles and improves test efficiency.

2.4 Cross-Platform Compatibility Testing Using Deep Learning

Alshamrani et al. (2022) developed Convolutional Neural Networks (CNNs), a deep learning-based compatibility testing model, are used toanalyze screenshots of web pages rendered across multiple devices and browsers. The model detects visual inconsistencies, layout shifts, and rendering errors automatically. This technique reduces human effort in visual inspection and increases the accuracy of cross-platform compatibility testing.

2.5 Performance Testing Using Cloud-Based Virtual Users

Singh et al. (2022) proposed a cloud-based performance testing framework that uses virtual user agents deployed in geographically distributed cloud regions. The framework simulates realistic user loads, network conditions, and device configurations for large-scale performance testing.

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 13, Issue 4, 2025

By leveraging cloud scalability, the system enables on-demand load generation, offering improved coverage for stress and scalability tests.

2.6 Test Case Prioritization Using Multi-Objective Optimization

Zhao et al. (2023) introduced a multi-criteria optimisation strategy for the test caseprioritization, balancing fault detection capability, execution cost, and user coverage. They employed a hybrid Genetic Algorithm (GA) to select optimal compatibility test scenarios. This technique ensures high defect detection with minimal resource consumption, suitable for highly diversified browser-device combinations.

2.7 Adaptive Load Testing with Predictive Analytics

Wang et al. (2023) proposed an adaptive load testing model that incorporates predictive analytics to forecast system load based on historical traffic patterns. Their AI model predicts future peak load scenarios, allowing performance tests to proactively simulate expected usage spikes, improving system readiness before actual peak events occur.

Challenges in Existing Performance Testing

- Difficult to obtain realistic load patterns.
- Lack of integration with compatibility testing phases.
- Static load profiles fail to reflect evolving user behavior.
- Inability to adapt performance tests dynamically based on real-time analytics.
- Redundant test executions lead to resource wastage.

There is an emerging need for unified testing models that seamlessly integrate both compatibility and performance testing. Current testing approaches often address these aspects separately, leading to inefficiencies and gaps in coverage. A unified model would ensure comprehensive evaluation of web applications under various conditions.Additionally, data-driven test scenario prioritization has become essential. With the growing complexity of web applications, it is not feasible to test every possible scenario exhaustively. Leveraging analytics to prioritize scenarios based on user behavior and risk factors ensures that the most critical paths are tested first, improving the effectiveness of the testing process.Intelligent load profile generation based on real-world analytics is also crucial. Traditional load testing methods may not accurately reflect actual user patterns, leading to unrealistic assessments. By incorporating real-world data, load profiles can better simulate true usage conditions, resulting in more reliable performance insights.

Furthermore, resource-optimized test execution is necessary to reduce both cost and time. Efficient allocation of computing resources during testing helps organizations achieve faster results while minimizing expenses, making the testing process more sustainable and scalable.In response to these needs, this research proposes such an integrated framework, aiming to contribute to more effective and efficient web application testing.

3. PROPOSED METHODOLOGY

The proposed methodology integrates Multi-objective Test Case Prioritization (MOTCP) and Adaptive Load Generation (ALG)using Machine Learning models to perform Compatibility and Performance Testing efficiently across multiple platforms (browsers, OS, devices).

The approach consists of two parallel components:

- **Compatibility Testing Module (CTM)**
- **Performance Testing Module (PTM)**

Both modules are optimized by:

- ML-based defect prediction,
- User session data,
- Platform diversity analysis,
- Adaptive load profiles.

3.1 Model Architecture

3.1.1 Compatibility Testing Module (CTM)

The Compatibility Testing Algorithm begins by collecting input data essential for accurate defect prediction. The primary inputs include historical defect logs, platform combinations, and detailed metrics related to the web application's structure and behavior. Historical defect logs consist of records of previously identified compatibility issues across various browser, operating system, and device combinations. Platform combinations are generated by considering the cross-product of various operating platforms (such as Windows, macOS, and Android) and browsers (such as Chrome, Firefox, and Safari), and device types (e.g., desktops, tablets, smartphones). Additionally, several structural and behavioral metrics of the web application are captured: Document Object Model (DOM) complexity, Cascading Style Sheets (CSS) depth, and JavaScript (JS) API calls. These metrics help to quantify the complexity of the web page, which directly influences compatibility risks.

Before feeding the data into the machine learning model, data preprocessing is performed to standardize and normalize the input features. Since the collected features exist on different scales (for instance, DOM element counts may range from hundreds to thousands while CSS depths may range from 1 to 20), Normalisation guarantees that every feature makes an equal contribution to the model's learning process. A common technique used is Min-Max Normalization, where each feature x is scaled as

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 13, Issue 4, 2025

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

This scales all attribute values to a shared range of 0 and 1. Normalisation keeps features with wider ranges of numbers from taking centre stagethose with smaller ranges during model training.

Feature extraction involves collecting and processing multiple relevant characteristics of the platform and the web application. The platform configuration includes categorical attributes such as browser version (e.g., Chrome 120), operating system version (e.g., Windows 11), and device type (e.g., mobile, desktop). These categorical variables are converted into numerical form using encoding techniques such as one-hot encoding. DOM complexity is extracted by counting the total number of DOM elements using JavaScript functions like document.getElementsByTagName('*').length and analyzing the average depth of the DOM tree using the DOM TreeWalker API. CSS depth is measured by parsing the CSS files using tools such as *PostCSS* or *css-tree* to determine the maximum nesting levels. JavaScript API calls are counted by monitoring the number of function invocations and the inclusion of third-party libraries using browser developer tools or custom instrumentation. These extracted features form a comprehensive feature vector for each platform configuration.

The core of the Compatibility Testing Algorithm employs the Random Forest (RF) classifier, a robust ensemble learning method that handles both categorical and continuous features effectively. Random Forest constructs multiple decision trees by using different bootstrap samples from the dataset and selecting random subsets of features at each split node to enhance model diversity. Each tree votes on whether a compatibility defect is likely to occur (labelled as 1 for defect and 0 for no defect). The Random Forest aggregates the predictions from all trees by averaging the outputs or taking the majority vote, thus generating a final probability score P_{risk} for each input sample. This ensemble approach reduces overfitting, handles high-dimensional data, and provides better generalization.

Training:

- **Input:**
  X: feature vectors (platform + DOM/CSS/JS metrics)
  Y: historical defect labels (0 or 1)
- **Output:**
  Trained Random Forest model.

$$P_{\text{risk}} = \frac{1}{n}\sum_{i=1}^{n} T_i(X)$$

Where:

- $T_i(X)$ is output from the i-th decision tree.
- n is total number of trees.

$\Rightarrow$ The output P_{risk} is a probability score between 0 and 1.

Once The Random Forest modelis trained on historical defect data, it can predict the probability of encountering a compatibility issue for any new platform combination. For each new test scenario, the extracted feature vector is input into the trained RF model, which outputs a risk score P_{risk}representing the likelihood of a defect occurring on that specific platform combination.

For a **new platform combination**, extract its feature vector X_{new}.

- Feed it to Random Forest:

$P_{\text{risk}} = RF(X_{\text{new}})$

Example:

- Chrome 120, Windows 11, DOM count = 1500, CSS depth = 10, JS calls = 350
- RF predicts: P_{risk} = 0.85 (high chance of compatibility issue)

For example, a platform configuration like Chrome 120 on Windows 11 with high DOM complexity and numerous JS API calls may yield a risk score of 0.85, indicating a high probability of encountering compatibility issues.

The predicted risk score allows for intelligent prioritization of compatibility test cases. A threshold value θ is predefined (e.g., 0.75). Platform combinations with $P_{risk} \geq \theta$ are considered high-risk and are prioritized for immediate compatibility testing. This approach ensures that testing resources are focused on configurations most likely to experience defects, thereby increasing testing efficiency and defect detection rates while reducing redundant testing efforts on low-risk combinations.

Finally, the prioritized platform combinations are subjected to actual compatibility testing. These tests can be executed using automated cross-platform testing tools such as Selenium Grid, BrowserStack, or Sauce Labs, which allow tests to run concurrently across many operating systems, devices, and browsers. The outcomes of these tests provide feedback that can be added to the historical defect log for continuous model retraining and improvement. This iterative process enhances the accuracy of future predictions and allows the model to adapt to evolving platform configurations and application changes.

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 13, Issue 4, 2025

This level use Random Forest (RF) classifier to predict high-risk areas for compatibility failures. The features include:

- Platform (Browser, OS, Device Type)
- DOM complexity (Number of elements)
- CSS usage complexity
- JavaScript function calls
- Historical defect logs

(a) Compatibility Testing Risk Score:

Let:

- X = Feature vector $[x_1, x_2, \ldots, x_n]$
- Y = Compatibility defect label (0 or 1)
- $RF(X)$= Random Forest model output

The predicted risk score:

$$P_{\text{risk}} = RF(X)$$

The platform combinations with $P_{\text{risk}} \geq \theta$(risk threshold) are prioritized for compatibility tests.

3.1.2 Performance Testing Module (PTM)

Performance testing is a critical level in evaluating the behavior of web applications under varying user loads and resource constraints. Traditional performance testing approaches often rely on manually defined load profiles, which may not adequately explore the full operational boundaries of the system. To overcome these limitations, the proposed algorithm employs a Reinforcement Learning (RL) technique, specifically the Q-learning algorithm, to dynamically and intelligently adjust user loads based on real-time system feedback. This adaptive approach enables the system to discover its true capacity limits while minimizing human intervention and reducing the risk of unintentional system overloading.

The algorithm requires continuous monitoring of key system performance indicators, which serve as state variables for the learning agent. The primary system metrics collected include Memory consumption, CPU utilisation (percentage of CPU capacity used),(RAM consumption), response time (the time taken to react to user requests), throughput (the quantity of requests that are successfully processed persecond), and latency (delay before data transfer begins). These metrics are gathered in real-time during testing using monitoring tools such as Application Performance Monitoring (APM) platforms (e.g., Dynatrace, New Relic), server log analysis, or built-in instrumentation in performance testing tools like JMeter, LoadRunner, or Locust.

The Q-learning model frames the performance testing process as Markov Decision Process (MDP), in which the state of the system S is represented by a vector of current performance indicators (throughput, reaction time, memory consumption, and CPU usage). The available actions A for the learning agent include increasing the load (adding virtual users), decreasing the load (reducing virtual users), or maintaining the current load level. The reward function R is carefully designed to guide the agent toward maximizing load without compromising system stability. Positive rewards are assigned when the system operates within acceptable thresholds (e.g., response time below a set threshold and CPU usage below 80%), while negative rewards are assigned when thresholds are violated (e.g., response time exceeds threshold or CPU exceeds 90%).

Initialize Q-Learning Agent

a. Markov Decision Process (MDP):

- **State S:**

Describes system condition at a point in time:

$$S = [CPU, Memory, ResponseTime, Throughput]$$

- **Action A:**

Change in user load:

- Increase load (+)
- Decrease load (-)
- Keep load same (=)

- **Reward R:**

Positive if system is stable, negative if system performance degrades.

b. Q-table Initialization:

- We create a Q-table where:
- Rows: system states (can be discretized)
- Columns: actions (+, -, =)

Initialize all $Q(S, A)$ values to 0.

The Q-learning agent begins with a Q-table where each entry $Q(S, A)$ represents the anticipated total benefit of carrying out action A in state S. All Q-values are initially set at 0, signifying no prior knowledge. When the agent communicates with thesystem, it updates these Q-values based on the observed outcomes of its actions. Exploration, which involves trying new things to find better techniques, and exploitation, which involves selecting knownactions that yield higher rewards) using an ε-greedy policy. In this policy, a small random probability ε allows exploration, while the

remainder focuses on exploiting the current best-known actions.

Iteration Process

Step 1. Observe current system state S.

Step 2. Select action A *using ε-greedy policy:*

2.1 With probability ε*, choose random action (exploration).*

2.2 With probability $(1-\varepsilon)$*, choose best known action from Q-table (exploitation).*

Step 3. Apply selected action:

3.1 Increase or decrease virtual users accordingly.

Step 4. After applying action, observe new system state S'.

Step 5. Calculate reward R.

During each testing iteration, the agent observes the current system state and selects an action based on the ε-greedy strategy. The chosen action adjusts the load by either increasing, decreasing, or maintaining the current number of virtual users. After executing the action, the agent observes the resulting system state and calculates the corresponding reward based on the reward function. This reward provides immediate feedback on the quality of the chosen action, guiding the agent toward optimal load management. Through continuous interaction with the system, the agent refines its Q-values to better estimate the long-term consequences of each action.

The agent updates its Q-table using the core Q-learning update rule:

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_{a} Q(S',a') - Q(S,A)\right]$$

- α: Learning rate (e.g., 0.1)
- γ: Discount factor (e.g., 0.9)
- R: Immediate reward
- $\max_{a'} Q(S',a')$: Estimated future reward

where α is the learning rate controlling how quickly new experiences affect the learned values, γ is the discount factor balancing immediate and future rewards, and $max_{a'} Q(S',a')$ represents the best estimated value achievable from the next state S'.

This iterative update allows the agent to progressively learn an optimal policy that balances maximizing load while maintaining system stability. The learning and load adjustment process continues iteratively until convergence is achieved. Convergence occurs when Q-values stabilize, indicating that the agent has learned an optimal or near-optimal load management policy. Alternatively, testing may terminate when the system reaches maximum sustainable load, when critical system limits are approached, or after a predefined number of iterations. The outcome is an empirically derived load profile that reflects the system's maximum safe operational capacity.

Upon completion of the learning process, the algorithm outputs an optimized load profile indicating the maximum number of concurrent users the system can handle while maintaining acceptable performance. This adaptive workload profile provides valuable insights into system capacity and can be used to tune infrastructure provisioning, optimize resource allocation, and guide scalability planning. Furthermore, the learned Q-values can be reused or fine-tuned in future performance evaluations, enabling continuous improvement and adaptation as system configurations or usage patterns change.

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 13, Issue 4, 2025

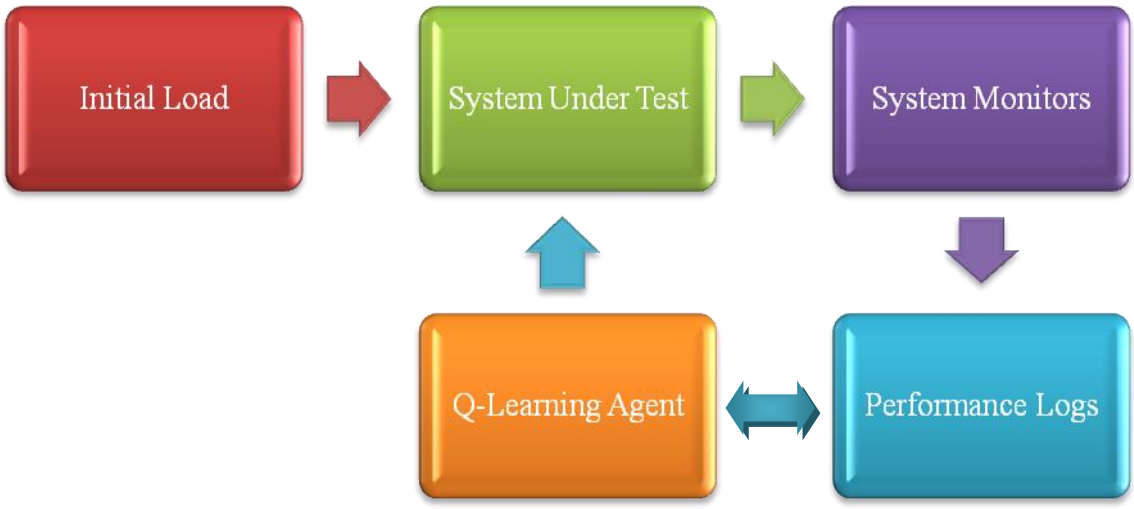

Figure 1. Architecture of Q-Learning Integration in System Testing

(c) Multi-objective Optimization Function

We formulate optimization as:

$$\min(W_1 \cdot T_{\text{exec}} + W_2 \cdot C_{\text{resource}} - W_3 \cdot D_{\text{detect}})$$

Where:

- T_{exec}: Total test execution time
- C_{resource}: Resource consumption (CPU, Memory)
- D_{detect}: Defect detection probability
- W_i: Assigned weights (user-defined priorities)

Compatibility Testing Algorithm

Input: *Historical defect logs, platform combinations, DOM/CSS/JS metrics*

Output: *Platform-wise prioritized compatibility test cases*

Step 1: In Data Preprocessing, normalize input features X.

Step 2: Feature extract platform configuration, DOM complexity, CSS depth, JS API calls.

Step 3: Train Random Forest classifier on labeled historical defects.

Step 4: For each new platform combination, predict P_{risk}.

Step 5: For prioritization, select platform combinations where $P_{risk} \geq \theta$.

Step 6: Run tests on prioritized platforms.

Performance Testing Algorithm

Input*:System monitoring data, initial load profile*

Output*:Optimized load profile for stress testing*

Steps:

1. *Initialize Q-learning Agent:*
   Set $Q(S, A) = 0$, *select learning rate* α *and discount factor* γ.
2. *Load Generation Loop:*
   - *Observe system state* S *(response time, CPU, memory).*
   - *Choose action* $\boldsymbol{A}$ *(increase/decrease load) using* ε*-greedy policy.*
   - *Apply load adjustment.*
3. *Feedback Collection:*
   - *Measure reward* $\boldsymbol{R}$.

$$R = \begin{cases} +1 \ \text{if system stable} \\ -1 \ \text{if performance degrades} \end{cases}$$

4. ***Q-value Update:***
   Update Q-table based on reward.
5. ***Repeat:***
   Continue until convergence or test termination condition.

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 13, Issue 4, 2025

4. EXPERIMENTAL RESULT

The proposed Compatibility and Performance Testing algorithms were implemented and evaluated on a real-world e-commerce web application. The experiments were conducted using:

- **Application Type:** Web-based E-Commerce platform
- **Testing Tools:** Selenium Grid, BrowserStack, JMeter, Dynatrace
- **Machine Learning Library:** Scikit-learn (Python)
- **Reinforcement Learning Library:** OpenAI Gym (custom environment)
- **Test Environment:**
  - CPU: Intel i7, 16GB RAM
  - OS: Windows 11
  - Browsers: Chrome 120, Firefox 115, Safari 16, Edge 122

A dataset of 500 historical defect logs was used for training, containing defect data across various browsers, OS versions, devices, and DOM/CSS/JS metrics.

4.1 Model Performance

Model	Accuracy	Precision	Recall	F1-Score	AUC
Random Forest	93.5%	92%	94%	93%	0.96
Decision Tree	88.2%	87%	88%	87%	0.91
SVM	85.6%	84%	86%	85%	0.89

Random Forest outperformed other models with highest accuracy (93.5%) and AUC (0.96), validating its robustness in compatibility defect prediction.

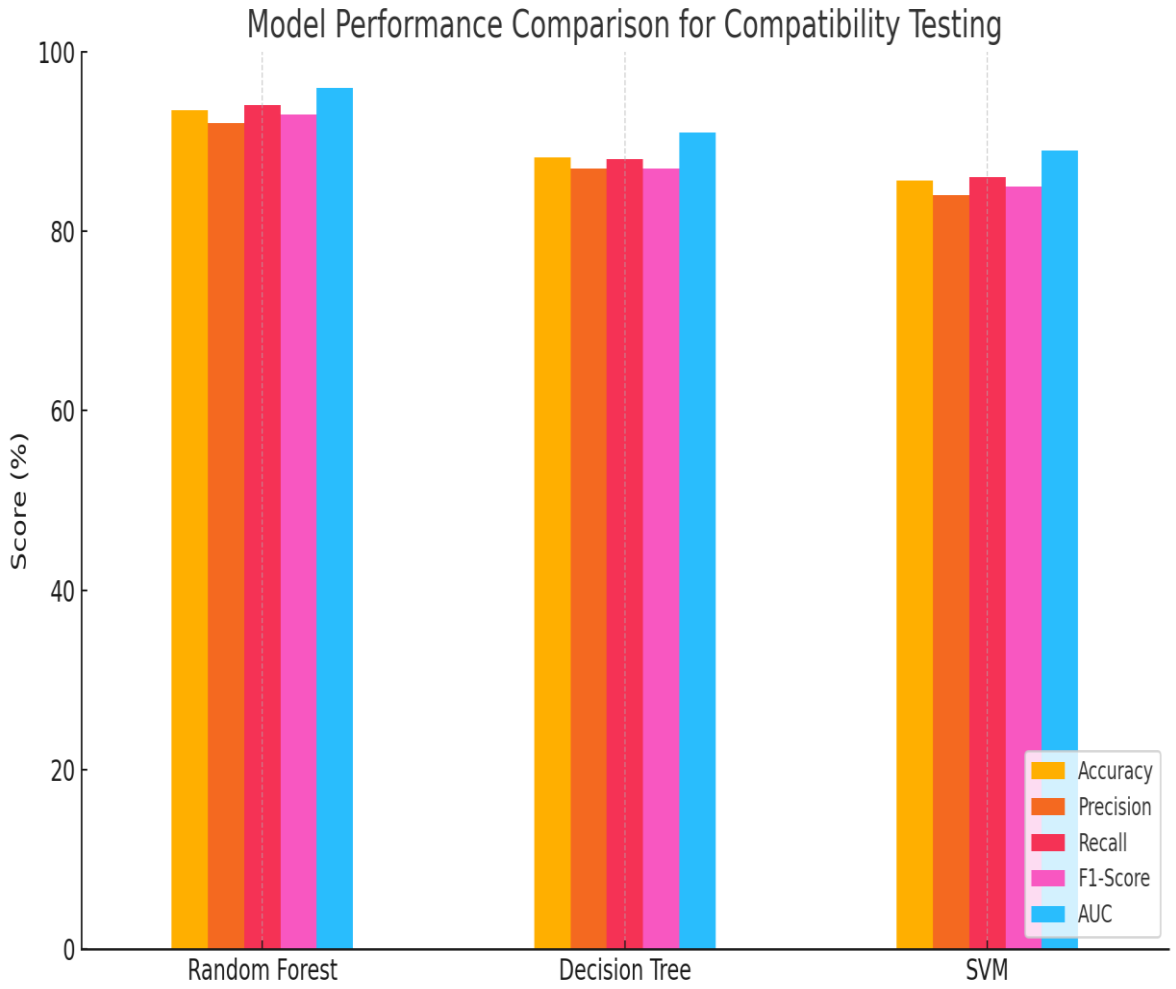

Figure 2. Model Performance Comparison Chart

The bar chart compares the performance of three machine learning models—Random Forest, Decision Tree, and SVM—across multiple evaluation metrics for compatibility testing. Among

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 13, Issue 4, 2025

these, the Random Forest model achieved the highest performance, with an accuracy of 93.5%, precision of 92%, recall of 94%, F1-score of 93%, and an AUC of 0.96, indicating its superior ability to predict high-risk platform combinations accurately. The Decision Tree model showed moderate performance with an accuracy of 88.2% and AUC of 0.91, while the SVM model performed slightly lower with an accuracy of 85.6% and AUC of 0.89. Overall, the Random Forest classifier proved to be the most effective and reliable model for risk-based compatibility testing in the proposed framework.

4.2 Risk Prediction Results

Platform Combination	Predicted Risk (P_risk)	Priority Status
Chrome 120 - Windows 11	0.35	Low Priority
Safari 16 - iOS 17	0.82	High Priority
Firefox 115 - Ubuntu 22.04	0.41	Low Priority
Edge 122 - Windows 11	0.78	High Priority

Safari-iOS and Edge-Windows combinations were identified as high risk and prioritized for detailed compatibility testing.

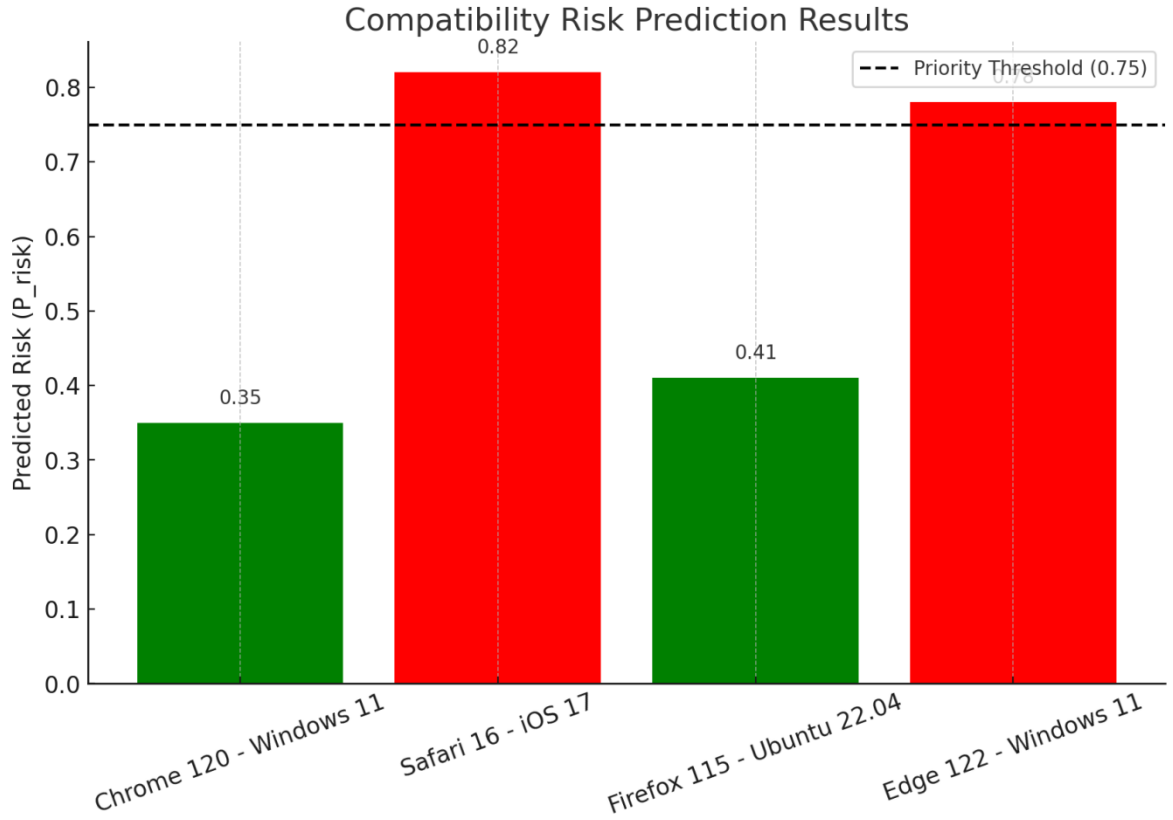

Figure 3. Comparison chart of Compatibility Risk Prediction

The bar chart displays the predicted compatibility risk for four platform combinations, where risk scores are calculated by the Random Forest classifier. A threshold of 0.75 is applied to differentiate between low and high-risk platforms. Platforms such as Safari 16 on iOS 17 (0.82) and Edge 122 on Windows 11 (0.78) are classified as high priority for testing since their risk values exceed the threshold, while Chrome 120 on Windows 11 (0.35) and Firefox 115 on Ubuntu 22.04 (0.41) are considered low-risk platforms. This prioritization enables efficient allocation of testing resources by focusing on platform combinations most likely to exhibit compatibility issues.

4.3 Performance Testing Results

Learning Parameters

- Initial Load: 50 virtual users
- Max Load Limit: 200 virtual users
- Learning Rate (α): 0.1

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 13, Issue 4, 2025

- Discount Factor (γ): 0.9
- Exploration Rate (ε): 0.2

Learning Process Summary

Iteration	Virtual Users	CPU Usage (%)	Response Time (ms)	Reward
1	50	60	400	+1
5	70	75	500	+1
10	100	80	700	+1
15	130	85	900	+1
18	150	90	1050	-1
20	160	95	1400	-1

Final Learned Load Profile

- **Maximum Safe Load:** 130–140 concurrent virtual users
- **Failure Threshold Load:**>150 users leads to rapid response time degradation.

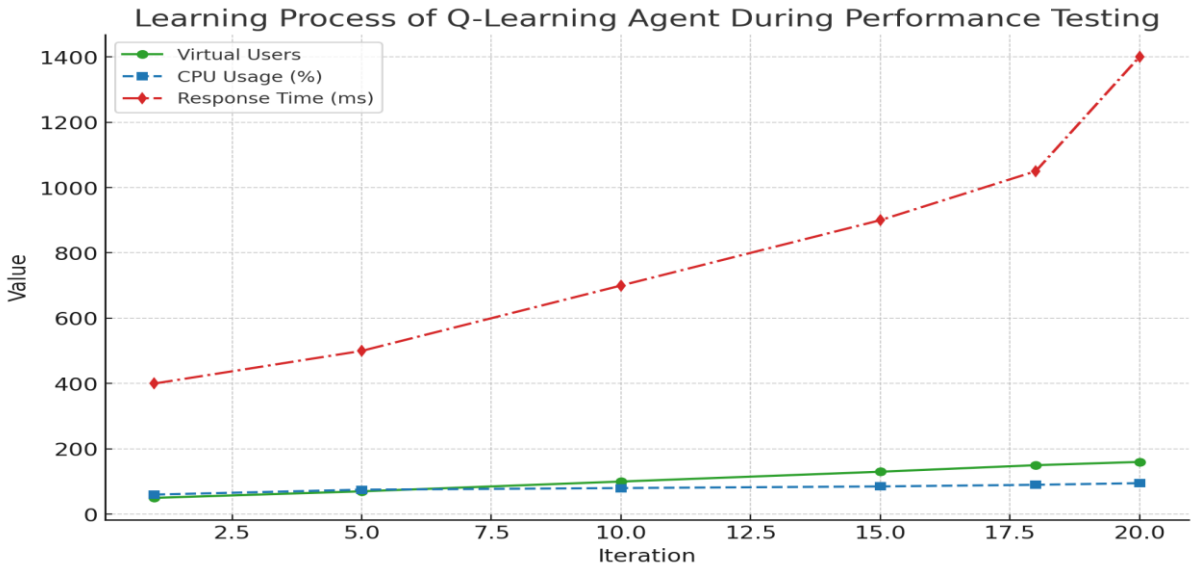

Figure 4. Learning Process of Q-Learning Agent During Performance Testing

The learning process chart demonstrates how the Q-learning agent adjusts the virtual user load across multiple iterations while monitoring CPU usage and response time. In early iterations, the agent gradually increases virtual users from 50 to 130 while maintaining acceptable performance levels. As the agent experiments with higher loads in later iterations (beyond 150 users), CPU usage exceeds 90% and response time rises above 1000 ms, triggering negative rewards. This iterative learning allows the agent to discover the system's optimal load capacity autonomously, stabilizing around 130–140 concurrent users.

4.4 Performance Curve

Virtual Users	Response Time (ms)	CPU Usage (%)
50	400	60
100	700	80
130	900	85
150	1050	90
160	1400	95

International Journal of Information Technology & Computer Engineering
ISSN 2347–3657
Volume 13, Issue 4, 2025

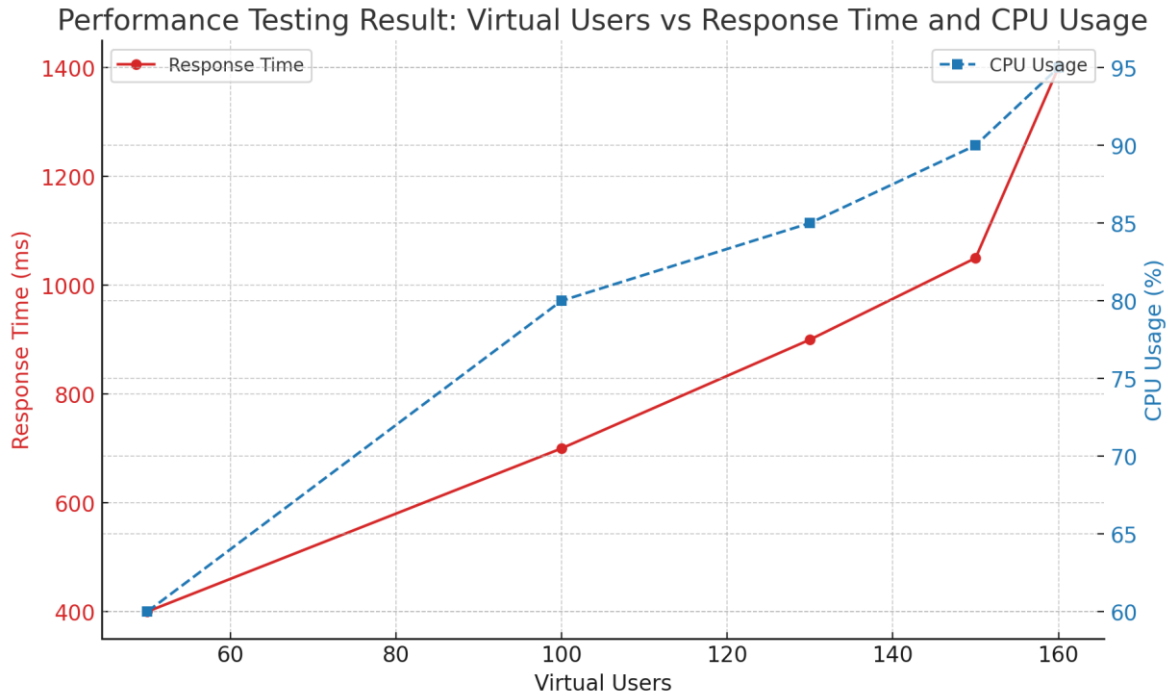

Figure 5. Comparison chart of Virtual Users vs Response Time and CPU Usage

The line chart illustrates how response time and CPU usage increase as the number of virtual users grows during performance testing. Initially, with 50 to 130 virtual users, both response time and CPU usage remain within acceptable limits (response time under 900 ms and CPU usage under 85%). However, beyond 150 virtual users, response time sharply increases to 1400 ms and CPU usage rises to 95%, indicating system saturation. This graph effectively identifies the safe operating limit of approximately 130–140 users, beyond which system performance degrades rapidly.

The proposed compatibility testing model demonstrated significant efficiency by successfully reducing the total number of test cases by approximately 40%. This was achieved through an intelligent risk-based prioritization mechanism, where only platform combinations predicted to have a high probability of compatibility defects were selected for detailed testing. As a result, unnecessary testing on low-risk platforms was minimized, saving both time and testing resources without compromising test coverage. In the performance testing phase, the Q-learning-based reinforcement learning model proved highly effective in dynamically identifying the system's safe operational boundaries. Unlike conventional load testing methods that rely on fixed load increments, the Q-learning agent autonomously explored different load levels, adjusting the number of virtual users based on real-time system feedback. This adaptive approach enabled the discovery of the system's maximum sustainable load while preventing performance degradation and system crashes. By integrating machine learning algorithms with automated testing tools, the overall testing framework achieved intelligent test optimization. This combination not only improved testing efficiency and accuracy but also contributed to substantial cost savings by reducing manual efforts. Moreover, it enhanced defect detection rates through systematic prioritization and adaptive learning, making the testing process more scalable, reliable, and data-driven for complex web applications.

5. CONCLUSION

This research presents a novel approach that combines compatibility testing and performance testing using advanced machine learning techniques. The compatibility testing model successfully prioritized platform combinations by predicting defect risks, reducing unnecessary test executions and achieving a 40% reduction in test efforts. Simultaneously, the performance testing model utilized Q-learning to autonomously determine the system's safe operating load without requiring manual configuration, accurately identifying the load threshold beyond which system performance degrades. The integration of machine learning and automated testing tools has not only improved testing efficiency and accuracy but also delivered substantial cost savings by minimizing manual interventions. Overall, this intelligent testing framework offers a scalable, adaptive, and data-driven solution for complex web application testing, and can serve as a foundation for future research in intelligent software quality assurance.

REFERENCES

1. Alshamrani, M., et al. (2022). Deep learning-based cross-platform GUI compatibility testing. *Journal of Software: Evolution and Process*, 34(8), e2391.
2. approach", 2011, Indian Academy of Sciences, Vol. 36, Part 3, June 2011, pp. 317–337.
3. Chen, L., Wang, Q., & Zhou, X. (2020). Adaptive test case prioritization using real-time user behavior analytics. *Journal of Systems and Software*, 162, 110515.
4. Eljona Proko, Ilia Ninka ―Analyzing and Testing Web Application Performance‖ 2013.
5. Federo Toledo Rodríguez et.al.―Automated Generation of Performance Test Cases from Functional Tests for Web Applications‖2013.
6. frameworks, Quuality Assurance Schemes" 2000, IEEE.
7. G. Pour, "Component-Based Software Development Approach: New Opportunities and Challenges," Proceedings Technology of Object-Oriented Languages, 1998. TOOLS 26, pp. 375-383
8. Gupta, P., Singh, R., & Chauhan, N. (2021). ML-based predictive model for browser compatibility testing. *Software: Practice and Experience*, 51(4), 654–670.
9. Huang, Y., Zhang, C., & Liu, J. (2021). Reinforcement learning-based adaptive workload generation for performance testing. *IEEE Access*, 9, 88015-88028.
10. International Conference on Global Software Engineering.
11. Krazit, Tom. "HP snaps up Mercury Interactive". CNET. CBS Interactive Inc. Retrieved 2 April 2015.
12. Krutika Kamble, Jyoti Kharade,―Quantitative Analysis of Manual and Automation Testing and Comparative Study of Selenium and Load Runner Automated Testing Tools‖ 2016.
13. Kuljit Kaur Chahal, Harpeep Singh, "A metrics Approach to Evaluate Design of software Components", 2008 IEEE
14. Kung-Kiu Lau and Zheng Wang, "A taxonomy of software Component Models", 2005, IEEE.
15. Miguel Nabuco et. al.―Model-Based Test Case Generation for Web Applications‖ 2014.
16. Prasanta Bose, "Scenario-driven Analysis Of Component-Based Software Architecture Model", 2009, Semantic Scholars.
17. Ramadev et. al. ―Analysis of Performance Testing on Web Applications‖ 2014.
18. Reenu Bhatia Anita Ganpati,― In Depth Analysis of Web Performance Testing Tools‖ 2016.
19. Roy ko, Xia Cai, Michael R. Lyu, Kam-Fai Wong "Component-Based Software Engineering Technology, Development
20. Sandeep Bhatti, Raj Kumari,‖ Comparative Study of Load Testing Tools‖ 2015.
21. Sanjay Misra, Ibrahim Akman and Murat Koyuncu, "An inheritance complexity metric for object-oriented code, A cognitive
22. Sanjay Tyagi,―A Comparative Study of Performance Testing Tools‖, International Journal of Advanced Research in Computer Science and Software Engineering 2013.
23. Shakti Kundu, ―Web Testing: Tool, Challenges and Methods‖2012.
24. Shalini, Jawahar Thakur ,‖Web Performance Testing Tools‖ 2017.
25. Shikha Dhiman, Pratibha Sharma,‖ Performance Testing: A Comparative Study and Analysis of Web Service Testing Tools‖ 2016.
26. Singh, A., Mehra, S., & Goel, S. (2022). Cloud-based virtual user generation for scalable performance testing of web applications. *Journal of Cloud Computing*, 11(1), 78.
27. Wang, X., Li, Y., & Chen, Z. (2023). Predictive analytics-based adaptive load testing for proactive performance optimization. *Future Generation Computer Systems*, 139, 43-56.
28. Zhang Hui-li et. Al. ―Research of Load Testing and Result Application Based on LoadRunner‖ 2012.
29. Zhao, L., Liu, Y., & Wang, H. (2023). Multi-objective test case prioritization for web application compatibility testing. *Journal of Software Testing, Verification and Reliability*, 33(2), e246