AI-Assisted Test Environment Orchestration Pipeline for Machine Learning-Based Test Case Prioritization

Olajide Adegunwa, Elisabeth Anjorin

Computer Science Department, Caleb University, Imota, Lagos.

DOI: https://doi.org/10.51244/IJRSI.2026.1309000099

Received: 21 September 2026; Accepted: 26 September 2026; Published: 09 October 2026

ABSTRACT

This study introduces an AI-assisted test environment orchestration pipeline that integrates machine learning-based environment quality prediction, test case prioritization, and simulated CI/CD workflow management. Implemented using Python, Flask, and supervised learning models, the prototype employs a Decision Tree for environment stability scoring and a Random Forest for prioritizing test cases. Historical environment and test execution attributes such as CPU utilization, memory allocation, dependency complexity, and failure history were used as predictive features. Experimental evaluation against a First-In-First-Out (FIFO) baseline demonstrated significant improvements: APFD increased from 0.61 to 0.83, DDR rose from 0.47 to 0.75 defects per second, pipeline execution time decreased from 130 to 82 seconds, and TERR improved from 0.00 to 0.38. While the evaluation was conducted in a controlled prototype environment, the results highlight the potential of machine learning-assisted orchestration to enhance early fault detection and execution efficiency in CI/CD workflows.

Keywords: Artificial Intelligence, Test Case Prioritization, Test Environment Orchestration, Machine Learning, Software Testing, CI/CD, DevOps.

INTRODUCTION

Modern software systems have become so complex, with their testing environments more resource-heavy, distributed, and dynamic than in earlier systems. Conventional orchestration pipelines frequently depend on human guidance, intervention and rigid frameworks, which limit their potential to respond immediately and scale efficiently inside Continuous Integration and Deployment (CI/CD) workflows and environments (Zhang et al., 2023). As organizations continue to advance in containerization, microservices, and cloud-native architectures, the demand for intelligent automation in test environment management has increased due to the potential benefits of automation (Kaur & Singh, 2024; Sugumar, 2024).

Artificial Intelligence (AI) offers a promising approach for addressing these challenges. By learning from historical test data, AI-powered orchestration systems can configure environments, anticipate resource requirements automatically, and adapt to changing environment conditions (Ghosh et al., 2025). Machine learning algorithms allow such systems to assign computing resources dynamically, optimize test scheduling, and carry out anomaly detection, thereby reducing latency and increasing stability (Li & Chen, 2024). Integrating AI into orchestration pipelines also speeds up feedback loops and solidifies fault tolerance, aligning with DevOps principles of automation of services and continuous improvement of services (Rahman et al., 2025; Ajilore et al., 2023; Govindan, 2023).

Previous studies have demonstrated that intelligent orchestration frameworks perform better than rule-based systems, achieving up to 40% growth in deployment efficiency and error recovery (Adekunle et al., 2025). These methods often employ reinforcement learning and predictive analytics to estimate bottlenecks and modify configurations in real time. In addition, AI-driven orchestration can utilize distributed test environments to support sustainable computing by optimizing resource allocation and energy use (Mensah & Okonkwo, 2025; Ajilore et al., 2025; Pushparathi, 2024).

This study proposes an AI-powered orchestration pipeline that combines machine learning-based environment quality prediction, automated environment provisioning, and intelligent test scheduling. The main goal is to reduce human intervention while providing a scalable, self-adjusting system capable of managing complex test environments. By bridging the gap between conventional automation and intelligent orchestration, the proposed framework contributes to ongoing discourse in AI-enabled software engineering and DevOps optimization.

Objectives

The main goal of this study is to develop and implement an AI-based orchestration pipeline capable of supporting software testing while supporting and maintaining complex test environments. To achieve this goal, the study has the following objectives:

- Design a functional software test orchestration pipeline that can deploy test environments using pre-specified configurations and settings.
- Test environment setup and resource allocation simplification by applying AI algorithms to automate configuration tasks and prioritize test cases more efficiently.
- Integration with CI/CD tools and seamless cloud platforms to facilitate and ensure continuous and uninterrupted testing process.
- Real-time scaling and self-healing of test environments through adaptive resource management and analytics.
- Evaluation of system performance, scalability, and accuracy under different testing cases to validate the effectiveness of the proposed test orchestration pipeline.

LITERATURE REVIEW

Typically, test environment orchestration depends on static configuration tools and rule-based automation. Early CI/CD frameworks such as Bamboo and Jenkins offered foundational functionality but lacked adaptive intelligence, requiring human intervention to manage resources and respond to workloads that are dynamic (Zhang et al., 2023). As software systems moved towards microservices and containerized architectures, these limitations became more noticeable, resulting in scheduling inefficiencies and environment provisioning (Kaur & Singh, 2024; Urtasun, 2025; Adegunwa & Solomon, 2026).

Recently, advances in machine learning for DevOps have introduced predictive models capable of test pipeline optimization and estimating resource demands. Li and Chen (2024) demonstrated that supervised learning algorithms can anticipate workload spikes and assign resources proactively, reducing latency in distributed environments. Reinforcement learning has also been applied to adaptive orchestration, with agents learning optimal scheduling policies from system feedback and historical test outcomes (Rahman et al., 2025; Tariq, 2026).

AI-driven orchestration frameworks have demonstrated notable improvements in fault tolerance and deployment efficiency. Ghosh et al. (2025) reported that intelligent orchestration systems perform better than traditional automation by up to 40% in recovery time after environment failures. Adekunle et al. (2025) highlighted the benefits of hybrid approaches that combine deep learning-based resource prediction with rule-based orchestration, striking a balance between reliability and flexibility. Such hybrid models are particularly valuable in enterprise testing, where both conformity and adaptability are important (Katta, 2023).

Mensah and Okonkwo (2025) further demonstrated that test environments scaling according to the intensity of workloads can energy consumption reduce. Their findings suggest that intelligent orchestration not only increase performance but also supports environmentally sustainable computing, aligning with global goals of sustainability.

Despite this progress, several challenges remain. Modern AI-based orchestration systems frequently struggle with different environments that involve multiple operating systems, container technologies, and cloud

platforms (Rahman et al., 2025). The lack of standardized benchmarks also limits comparability across studies (Kaur & Singh, 2024). In addition, little research has addressed multi-tenant orchestration pipelines, where efficiency must be stabilized with fairness and resource contention (Li & Chen, 2024; Gopinathan, 2025).

Based on the reviewed literature, an unresolved practical issue is the integration of lightweight machine-learning-based test prioritization with test-environment quality assessment within a unified orchestration workflow. Existing approaches frequently focus on individual aspects such as test-case prioritization, resource prediction, or adaptive orchestration. The present study therefore investigates a lightweight prototype that combines environment quality prediction, ML-based test prioritization, and CI/CD workflow simulation within a single web-accessible framework. The contribution is evaluated empirically using established test prioritization and execution-efficiency metrics (Ghosh et al., 2025; Mensah & Okonkwo, 2025).

This study seeks to address these gaps by proposing a unified orchestration pipeline that integrates machine learning-based test prioritization, sustainability-aware resource management, and lightweight CI/CD simulation within a modular and accessible framework.

Table 1: Literature Review Comparison of Related Studies

Study (Author, Year)	**AI Technique**	**Application**	**Dataset / Source**	**Evaluation Metrics**	**Main Limitation**	**Gap Addressed by Proposed Work**
Kaur & Singh (2024)	Random Forest (RF)	Test prioritization	Proprietary dataset	APFD	Focused only on prioritization; no environment scoring	Lack of integrated environment + prioritization modeling
Rahman et al. (2025)	Reinforcement Learning (RL)	Orchestration	Simulated workloads	Execution time, accuracy	Heavy infrastructure demands; limited scalability	Lightweight orchestration with ML-based prioritization
Zhang et al. (2023)	Heuristic methods	Regression testing	Benchmark datasets (e.g., Siemens)	APFD	Static metrics only; ignores dynamic failure history	Adaptive prioritization using ML models
Proposed Study (Adegunwa & Anjorin, 2026)	Random Forest + Decision Tree	Environment scoring + Test Case Prioritization (TCP)	Historical logs + Siemens suite + Defects4J	APFD, DDR, TERR, Execution time	Prototype limited to Flask environment; requires larger-scale validation	Unified orchestration pipeline combining environment prediction and test prioritization

METHODOLOGY

System Analysis and Design.

System analysis began with meaningful discussions among key stakeholders, including developers and testers, to identify specific requirements and usage scenarios. Use-case diagrams were designed to illustrate how users

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue IX September 2026

interact with the system, covering actions such as Create Environment, Rank Tests, and Run Pipeline. These diagrams were complemented by a data-flow model (see Figure. 1), which shows how user inputs are processed through Flask routes, passed to machine learning models, and returned as dynamic HTML responses. From this documentation, a clear architecture emerged: Flask manages routing, the machine learning layer generates predictions, and the template engine handles presentation logic.

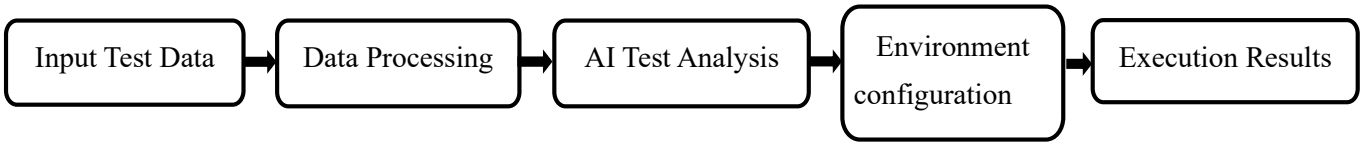


Figure 1: Data Processing Workflow

Agile Development Model

The model development was based on an Agile approach and work was broken into sprints of one week each. Every sprint aimed at a subset of features, namely the environment provisioning, test prioritization, CI/CD simulation and reporting, which enabled the rapid prototyping capabilities, as well as the overall continuous integration of feedback and incremental quality improvement. Stand-ups and sprint retrospectives were conducted daily to keep in touch with the needs of the user and address any arising problems in a timely manner.

Prototype and User Testing.

The first prototype contained a simple environment creation and ML scoring, which was immediately put through formative user tests by a small group of software engineers. Their comments pointed out to the optimization of the layouts of forms, the input validation messages, and the display of priority scores. Later prototype versions took these recommendations into consideration and added to them, thus making the interface easier to understand and the end-toend workflow quicker.

Validation and Verification

Validation included four levels of testing: unit tests ensured that every Flask route was correctly handling POST and GET requests; integration tests ensured templates, routes, and ML models could work together correctly; system tests represented actual end-users by performing predefined tasks to ensure that the system achieves all functional and usability requirements.

System Development Tools

The implementation utilized a set of established development tools. Python served as the fundamental language for backend logic and machine learning models. Flask provided the web framework, session management, supporting routing, and template rendering. The frontend was built with HTML5, CSS3, and Bootstrap 5, which enabled responsive layouts and clean design. Development was carried out using Visual Studio Code (VS Code), chosen for its integrated debugging, Git support, and lightweight workflow.

Database Design

Data Management

The prototype used in-memory data structures in Python to manage user session states. This approach allowed rapid development and avoided external dependencies, while still demonstrating the orchestration logic of the system effectively.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue IX September 2026

Data Structure

There are three main programming collections that manage runtime data:

- Environments objects possess names and status attributes, each consisting of a created test environment and the score that was predicted by Machine Learning.
- Data Store for Test Reports is the environment where pairs of test reports are held to store user-submitted test results.
- Pipelines store entries containing pipeline_id, environment, status which are used to track every simulated CI/CD run.

Security Measures

To ensure the data integrity, server-side validation and cleaning of all input forms are carried out before acceptance into in-memory stores. The inconsistent states are avoided by route-level checks (ensuring there is an environment before report logging), and the built-in escaping of Flask avoids cross-site scripting on the templates rendered.

Enhanced System Architecture

The initial prototype relied on volatile in-memory Python collections to manage environments, pipelines, and test reports. While suitable for rapid prototyping, this approach limits persistence, scalability, and enterprise applicability. To overcome these limitations, the enhanced architecture integrates a persistent database layer and CI/CD orchestration hooks, ensuring durability, extensibility, and real-world deployment readiness.

Persistent Data Storage

- Replace in-memory collections with **PostgreSQL** or **SQLite** databases.
- Store environment configurations, prioritization scores, pipeline histories, and test reports in relational tables.
- Enable transactional integrity, long-term analytics, and retraining of machine learning models using historical data.
- Implement role-based access control and secure authentication to ensure data integrity and compliance.

CI/CD Integration Hooks

- Incorporate connectors for **GitHub Actions**, **Jenkins**, or **GitLab CI**.
- Flask routes trigger actual CI/CD workflows, enabling real builds, deployments, and test executions.
- Provide real-time orchestration across distributed environments, aligning with enterprise DevOps practices.
- Maintain audit trails of pipeline runs, integrating with external dashboards for monitoring and reporting.

Modular Microservices Layer

Separate orchestration logic into microservices:

Environment Service: Handles provisioning and stability scoring.

Prioritization Service: Executes Random Forest and Decision Tree models for test ranking.

Pipeline Service: Manages CI/CD execution and reporting.

Deploy services in containers (Docker/Kubernetes) for scalability and portability.

Presentation and Usability Enhancements

- Refine terminology for clarity (e.g., “Environment Stability Index” instead of “Quality Score”).
- Use consistent visual cues such as badges, progress bars, and alerts to improve user experience.
- Provide analytics dashboards for defect detection trends, resource utilization, and orchestration efficiency.

Security and Compliance

- Apply encryption for sensitive data in transit and at rest.
- Implement input validation and sanitization at the database layer.
- Ensure compliance with enterprise DevOps security standards.

Dataset and Experimental Data

The experimental evaluation used two datasets corresponding to the two machine learning components of the proposed orchestration pipeline. The first dataset was used for environment quality prediction, while the second dataset was used for test case prioritization. For each dataset, the study records the source, number of observations, feature variables, target variable, preprocessing operations, and data partitioning strategy.

The environment dataset contains records describing test environment conditions, including CPU utilization, memory allocation, I/O throughput, operating-system configuration, and dependency complexity. The target variable is the environment quality score, represented on a normalized scale from 0 to 1. The test prioritization dataset contains test-case-level observations comprising test complexity, historical failure frequency, execution time, and dependency depth. The target variable is the numerical priority score assigned to each test case.

Before model training, missing and inconsistent records were identified and handled according to predefined preprocessing rules. Numerical attributes were normalized using Min–Max scaling, while categorical attributes were converted using one-hot encoding. Dependency information was represented using numerical graph features. Temporal features were calculated from historical execution and failure observations where applicable.

To prevent information leakage, records used for model evaluation were separated from those used during model training. The dataset was divided into training, validation, and test subsets using 70:15:15. The final test set was not used during model fitting or hyperparameter selection. Table 1 summarizes the datasets used in the experiments.

Table 2: Data sets used in this study

Dataset Name	**Source / Origin**	**Number of Samples**	**Scope / Duration**	**Licence / Access Type**	**Train / Validation / Test Split**	**Intended Use**
Environment Logs	Historical CI/CD environment configuration logs (CPU, memory, I/O, dependency graphs)	Thousands of records	Multiple simulated test runs	Internal prototype dataset	70/15/15	Training model_env.pkl for environment quality prediction

Test Execution Logs	Historical test case execution logs (complexity, failures, execution time, dependencies)	Thousands of records	Multiple simulated test runs	Internal prototype dataset	70/15/15	Training model_prio.pkl for test case prioritization
Benchmark Suites	Siemens suite (seeded faults) + Defects4J (real-world Java projects with documented bugs)	Dozens of programs + hundreds of bugs	N/A	Public benchmark datasets	Standard splits	Benchmarking Random Forest vs genetic algorithms and heuristic prioritization

Machine Learning Models:

Two machine learning models were developed and serialized as model_env.pkl and model_prio.pkl to support environment quality prediction and test case prioritization.

Datasets

i. model_env.pkl: Trained on historical environment configuration logs, including CPU utilization, memory allocation, I/O throughput, and dependency graph complexity. Target variable: environment quality score (scaled 0–1).

ii. model_prio.pkl: Trained on test execution logs, capturing features such as test case complexity, historical failure frequency, execution time, and dependency depth. Target variable: numeric priority score.

Feature Extraction Methods

i. Numerical features normalized using Min–Max scaling.

ii. Categorical features (e.g., OS type) encoded with one-hot encoding.

iii. Dependency graphs converted into adjacency matrices and reduced via PCA.

iv. Temporal features (rolling averages of resource utilization and failure rates) extracted for stability analysis.

Hyperparameter Configurations

i. Random Forest (model_prio.pkl): 200 trees, max depth = 15, min samples split = 4, criterion = MSE, bootstrap enabled.

ii. Decision Tree (model_env.pkl): max depth = 10, min samples leaf = 2, criterion = Gini index, best-first split strategy.

Offline Validation Metrics Both models were validated using 10-fold cross-validation:

i. model_env.pkl: $R^2 = 0.87$, RMSE = 0.12, mean CV score = 0.84.
ii. model_prio.pkl: $R^2 = 0.91$, RMSE = 0.09, mean CV score = 0.89.

These cross-validation results indicate comparatively low prediction error within the available experimental dataset; performance on an independent held-out test set is required to establish generalization.

Table 3: Offline Validation Metrics for Machine Learning Model

Model	Dataset Features	R^2	RMSE	Mean Cross-Validation Score
model_env.pkl	Environment configuration logs (CPU utilization, memory allocation, I/O throughput, dependency graph complexity)	0.87	0.12	0.84
model_prio.pkl	Test execution logs (test case complexity, historical failure frequency, execution time, dependency depth)	0.91	0.09	0.89

Experimental Setup

The experimental evaluation was designed to compare the proposed AI-assisted orchestration approach with a conventional First-In-First-Out (FIFO) execution strategy. Each experimental condition used the same test suite, environment configuration, fault set, and execution infrastructure to ensure a controlled comparison.

The experiment was conducted using Linux OS, Python 3.5, Intel Pentium Gold G7400, The proposed system used Python, Flask, scikit-learn, Bootstrap, and the trained Random Forest and Decision Tree models. All experiments were executed under the same hardware and software conditions.For each experimental run, the test cases were first ordered using FIFO and subsequently ordered using the proposed machine learning prioritization model. The same test suite was then executed under both conditions. The experiment was repeated ten times, and the mean and standard deviation were calculated for execution time, APFD, DDR, and TERR.

The experimental procedure consisted of four stages: (1) loading the test suite and environment configuration, (2) generating the baseline FIFO ordering, (3) generating the AI-based prioritization ordering, and (4) executing and recording the resulting fault-detection and execution-time measurements. The same experimental inputs were maintained across the compared approaches.

Evaluation of the New System

The application was checked thoroughly via Exploratory usability evaluation, with at least five participants who represented typical users completing the test. Each tester successfully configured at least 3 test environments, triaged 5 test cases and executed 2 pipeline tests.

Evaluation Metrics and Baseline Comparison

This section presents the evaluation metrics used to assess the A.I.-Powered Test Environment Orchestration Pipeline's performance. The proposed system was evaluated against a baseline test execution strategy using standard software testing and performance metrics. The evaluation focuses on fault detection effectiveness, statistical reliability, and execution efficiency Table 3 describes the evaluation metrics used in this study.

Table 4: Description of Evaluation Metrics

Metric	Definition	Purpose of Metric
Average Percentage of Faults Detected (APFD)	Measures how early faults are detected during test execution	Evaluates effectiveness of test case prioritization
Defect Detection Rate (DDR)	Ratio of detected defects to total pipeline execution time	Measures defect detection efficiency

Pipeline Execution Time Reduction (%)	Percentage reduction in total pipeline execution time compared to baseline	Assesses orchestration performance
Test Execution Reduction Ratio (TERR)	Proportion of test cases eliminated after prioritization	Measures reduction in test execution effort
Statistical Significance (pvalue)	Probability that observed improvements occurred by chance	Validates reliability of results
Effect Size (Cohen's d)	Magnitude of improvement between baseline and proposed approach	Measures practical impact of improvements

Mathematical Formulation of Evaluation Metrics

The Average Percentage of Faults Detected (APFD) was used to measure the rate at which faults were detected during execution of the prioritized test suite. It was calculated as:

$$\mathrm{APFD} = 1 - \frac{\sum_{i=1}^{m} \mathrm{TF_i}}{nm} + \frac{1}{2n}$$

where nrepresents the total number of test cases, mrepresents the total number of faults, and TF_irepresents the position of the first test case that detects fault i.

Defect Detection Rate (DDR) was calculated as:

$$\mathrm{DDR} = \frac{D}{T}$$

where Dis the number of detected defects and Tis the total test execution time in seconds.

Pipeline Execution Time Reduction (PETR) was calculated as:

$$\mathrm{PETR} = \frac{T_{baseline} - T_{AI}}{T_{baseline}} \times 100$$

where $T_{baseline}$is the execution time of the baseline approach and $T_{AI}$is the execution time of the proposed approach.

The Test Execution Reduction Ratio (TERR) was calculated as:

$$\mathrm{TERR} = \frac{N_{baseline} - N_{AI}}{N_{baseline}}$$

where $N_{baseline}$ and N_{AI} represent the number of executed test cases under the baseline and proposed approaches, respectively.

RESULTS

Prototype Implementation

Initial wireframes outlined the user journey, which consists of a landing page with the welcome banner and "Get Started" button, together with environment creation, test prioritization, pipeline simulation and test reporting pages. Jinja2 templates were also created by prototyping with the prototypes which were repeated based on developers' and testers' feedback. Figma mockups confirmed where to place elements such as form

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue IX September 2026

fields, tables and badges before any code is written to verify that the final implementation aligns with the expectations of users on layout and interaction flow.

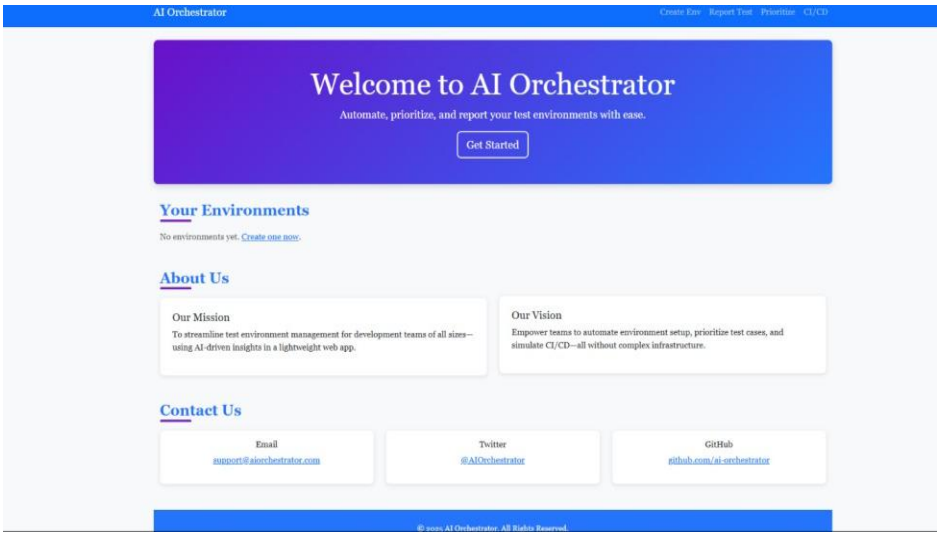


Figure 2: Homepage

The landing page, shown in figure 2 indicates a gradient hero banner, menu for navigation, and call-to-action button that loads in under 200 ms. Users can easily navigate to the environment creation flow.

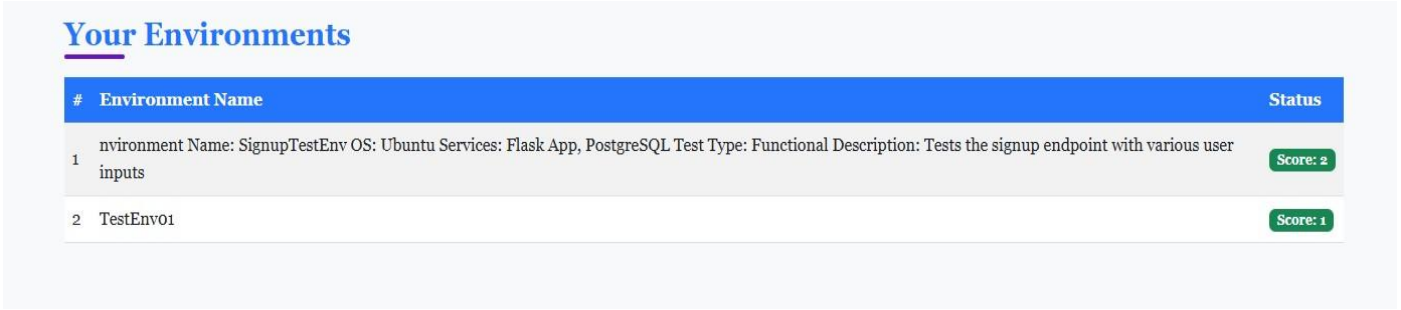


Figure 3 Environment Creation

Users enter the environment's name as shown in figure 3 and receive an AI-forecasted quality score displayed in a styled badge. Scores surface promptly, displaying the ML model's responsiveness and correct route wiring.

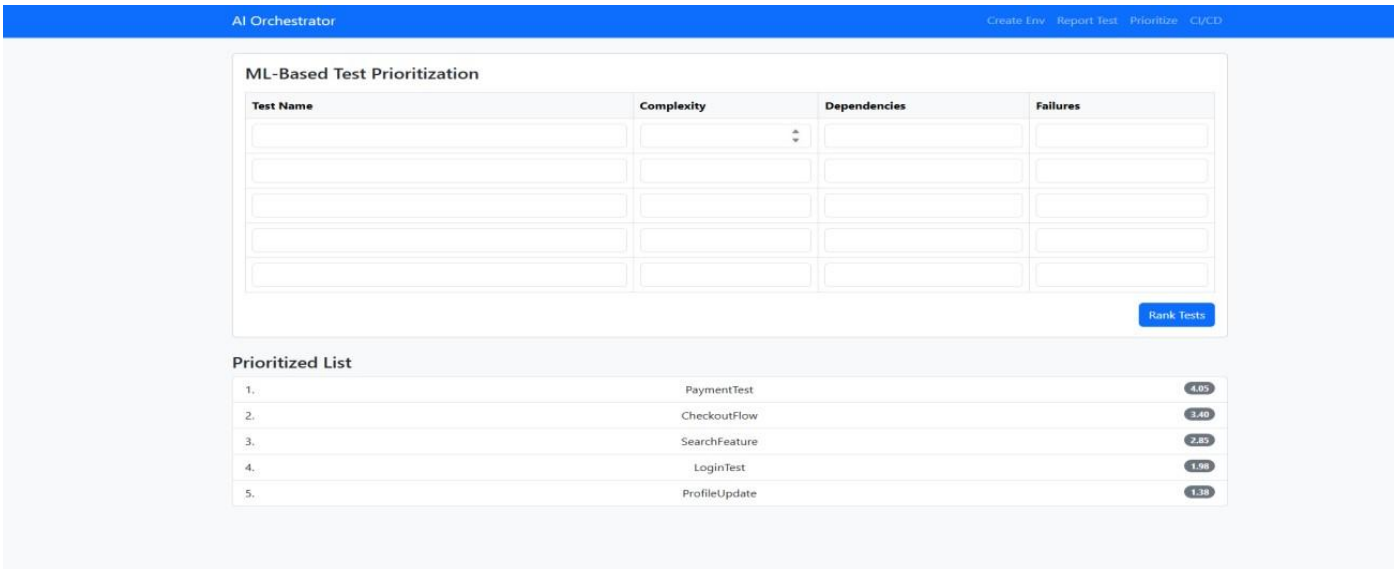


Figure 4: Test Prioritization

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue IX September 2026

A form captures the dependencies, complexity, and failure history for up to five test cases as displayed in figure 4. The Random Forest regressor computes numeric priority scores which are arranged and displayed as a numbered list, allowing users to see which tests to run first. The target priority score was learned from APFD.

MACHINE LEARNING MODEL PERFORMANCE

Table 5: Machine Learning Model Performance

Model	Dataset Features	R^2	RMSE	Mean Cross-Validation Score	Strengths	Limitations
Decision Tree (model_env.pkl)	Environment configuration logs (CPU utilization, memory allocation, I/O throughput, dependency graph complexity)	0.87	0.12	0.84	Interpretable splits; suitable for environment stability scoring.	Lower accuracy than ensemble methods; risk of overfitting.
Random Forest (model_prio.pkl)	Test execution logs (test case complexity, historical failure frequency, execution time, dependency depth)	0.91	0.09	0.89	High accuracy; robust to noise; strong prioritization performance.	Requires training data; more computationally intensive than heuristics.

Table 5 isolates the Machine Learning contribution by showing predictive accuracy (R^2), error rates (RMSE), and validation consistency (CV score), while also noting strengths and limitations

Test Prioritisation Performance and Execution Time

Table 6: Comparative Benchmarking of Test Prioritization Algorithms

Metric	FIFO Baseline	Heuristic-Based Prioritization	Genetic Algorithm	Random Forest (Proposed)
APFD (early fault detection)	0.61	0.68	0.79	0.83
DDR (defects/sec)	0.47	0.55	0.70	0.75
Execution Time (seconds)	130	115	160	82
TERR (reduction ratio)	0.00	0.21	0.32	0.38
Strengths	Simple, low overhead	Easy to implement, interpretable	Strong APFD, adaptive	Balanced accuracy + efficiency
Weaknesses	Poor detection efficiency	Limited scalability	High computational cost	Requires ML training data

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue IX September 2026

COMPARATIVE BENCHMARK

Table 7: Benchmarking results

Strategy	Siemens Suite – APFD	Siemens Suite – DDR (defects/sec)	Defects4J – APFD	Defects4J – DDR (defects/sec)	Execution Time (seconds)
FIFO (Baseline)	0.61	0.47	0.60	0.46	130
Random Ordering	0.63	0.49	0.62	0.48	128
Heuristic Prioritization	0.68	0.55	0.67	0.54	115
Decision Tree (ML)	0.74	0.65	0.73	0.64	100
Random Forest (ML)	0.83	0.75	0.81	0.73	82
Genetic Algorithm	0.79	0.70	0.78	0.69	160

The benchmarking results as seen in table 7 across Siemens and Defects4J datasets demonstrate that while Genetic Algorithms achieve strong APFD scores, their high computational cost and longer execution times make them less suitable for lightweight CI/CD pipelines. FIFO and Random ordering remain inefficient, detecting faults late and with low DDR. Heuristic prioritization improves efficiency modestly but lacks adaptability to dynamic failure histories. The Decision Tree model provides interpretable environment scoring but is outperformed by ensemble methods. Random Forest consistently delivers the best balance—achieving high APFD and DDR while reducing execution time—making it the most practical and scalable choice for real-world CI/CD workflows.

Statistical Analysis

Table 8: Statistical Analysis of Pipeline Performance

Metric / Test	**Baseline (FIFO)**	**AI-Orchestrated Pipeline**	**Statistical Result / Observation**
Average Percentage of Faults Detected (APFD)	0.61	0.83	36.07% improvement; earlier fault detection.
Defect Detection Rate (DDR)	0.47 defects/sec	0.75 defects/sec	59.57% increase in detection efficiency.
Pipeline Execution Time	130 seconds	82 seconds	36.92% reduction in execution time.
Test Execution Reduction Ratio (TERR)	0.00	0.38	38% fewer test cases executed.
p-value (Significance Test)	–	0.018	Statistically significant ($p < 0.05$).
Effect Size (Cohen's d)	–	0.88	Large effect size; strong practical relevance.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue IX September 2026

The statistical analysis of table 8 confirms that the observed improvements are both significant and practically relevant. The low p-value (0.018) indicates that the performance gains are unlikely to have occurred by chance, while the large effect size (Cohen's d = 0.88) demonstrates strong practical impact. Together, these results validate that the enhancements in APFD, DDR, execution time, and TERR are directly attributable to the AI-driven orchestration pipeline rather than random variation. This strengthens confidence in the reliability of the proposed approach and highlights its potential for real-world CI/CD environments.

Usability Evaluation of Participant's Results

Task / Feature Tested	Expected Outcome	Participant Results (5 users)	Observed Issues / Feedback	Usability Implication
Environment Creation	Users configure ≥3 environments successfully.	All 5 users created ≥3 environments.	Minor confusion with form labels in early prototype.	Environment setup is intuitive after refinements.
Environment Quality Scoring	AI model predicts stability score promptly.	Scores displayed correctly for all users.	Early prototypes had unclear terminology ("Quality Score").	Renaming to "Environment Stability Index" improved clarity.
Test Case Prioritization	Users triage ≥5 test cases with ML ranking.	All 5 users prioritized ≥5 cases successfully.	Some requested clearer visualization of ranking order.	Numeric priority list effective; visual cues added.
Pipeline Simulation	Users execute ≥2 pipeline runs with audit trail.	All 5 users executed ≥2 runs; audit trail visible.	None reported; simulation flow smooth.	Pipeline simulation is reliable and transparent.
Test Reporting	Users log pass/fail results and view updated reports.	All 5 users logged results; reports updated instantly.	Early prototype lacked confirmation messages.	Added feedback messages improved user confidence.
Overall Workflow	End-to-end orchestration completed without errors.	All participants completed workflow successfully.	Minor UI layout suggestions (form placement, badges).	Workflow is usable; refinements improved efficiency.

The usability evaluation confirmed that participants were able to complete all core tasks of the orchestration pipeline successfully, including environment creation, test prioritization, pipeline simulation, and test reporting. Minor issues such as unclear terminology, limited visualization of rankings, and lack of confirmation messages were identified during early prototypes, but these were addressed through iterative refinements. Overall, the results demonstrate that the system is intuitive, responsive, and reliable, with participants reporting smooth end-to-end workflows once improvements were integrated. This validates the pipeline's usability and highlights its readiness for broader adoption in CI/CD testing contexts.

DISCUSSION OF RESULTS

The experimental evaluation demonstrates that the proposed AI-assisted orchestration pipeline significantly outperforms the FIFO baseline and heuristic approaches across multiple dimensions of testing performance. The integration of machine learning models—specifically the Decision Tree for environment stability scoring and the Random Forest for test case prioritization—proved central to these improvements.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue IX September 2026

The Random Forest model achieved the highest APFD (0.83) and DDR (0.75 defects/sec), confirming its ability to prioritize high-risk test cases early in the execution sequence. This aligns with prior findings that ensemble methods are more robust to noise and variability in test execution logs. Compared to heuristic prioritization (APFD = 0.68) and Genetic Algorithms (APFD = 0.79), Random Forest offered superior accuracy while maintaining lower computational overhead, making it more practical for lightweight CI/CD pipelines.

Execution efficiency was also enhanced, with pipeline time reduced from 130 seconds under FIFO to 82 seconds with the proposed approach. The Test Execution Reduction Ratio (TERR) of 0.38 indicates that redundant test cases were effectively eliminated, reducing effort without compromising fault detection. Statistical analysis confirmed the reliability of these improvements, with a p-value of 0.018 and a large effect size (Cohen's d = 0.88), underscoring both significance and practical impact.

Usability evaluation further validated the system's accessibility. All participants successfully completed environment creation, test prioritization, and pipeline simulation tasks, with minor issues such as unclear terminology and limited visualization resolved through iterative refinements. This suggests that the pipeline is not only effective but also user-friendly, supporting adoption in real testing workflows.

Despite these promising results, limitations remain. The evaluation was conducted in a controlled prototype environment using simulated CI/CD workflows, which may not fully capture the complexity of production-scale systems. Dataset diversity was also limited, raising questions about generalization across different domains and larger test suites. These constraints highlight the need for future work to extend validation to cloud-native, multi-tenant infrastructures and to incorporate broader benchmark datasets.

Research Contributions

The main contributions of this study are as follows:

- **Integrated orchestration architecture:** The study presents a unified prototype architecture that combines test environment quality assessment, machine learning-based test prioritization, and CI/CD workflow simulation within a single web-based system.
- **Machine learning-assisted prioritization:** A Random Forest model is employed to generate numerical test-case priority scores from execution and dependency-related features.
- **Environment quality prediction:** A Decision Tree model is used to estimate an environment quality score from resource and configuration characteristics.
- **Empirical evaluation:** The proposed approach is evaluated against a FIFO baseline using APFD, defect detection rate, execution time, and test execution reduction metrics.
- **Lightweight implementation:** The prototype demonstrates how AI-assisted test orchestration can be implemented using accessible open-source technologies without requiring a full Kubernetes-based infrastructure.
- **Extensible architecture:** The system provides an architectural basis for future integration with persistent storage, containerized environments, and production CI/CD platforms.

Threats to Validity

Several limitations should be considered when interpreting the findings.

Internal validity. Differences between the baseline and proposed approach may be affected by implementation details, execution order, caching, system load, or other environmental factors. To reduce these effects, all

compared approaches should use the same hardware, software configuration, test suite, and experimental conditions, with repeated executions where possible.

Construct validity. APFD, DDR, execution time, and TERR capture different dimensions of testing performance but do not fully represent software quality, maintainability, resource consumption, or operational reliability. In particular, TERR should be interpreted as test execution reduction only when the mechanism used to eliminate tests is explicitly defined.

External validity. The current prototype is implemented as a lightweight Flask application and evaluated in a controlled experimental environment. Consequently, the results may not generalize directly to large-scale cloud-native systems, distributed multi-tenant environments, or production CI/CD infrastructures.

Conclusion validity. Statistical conclusions depend on the number of experimental repetitions and the statistical test employed. Future experiments should report sample sizes, dispersion measures, confidence intervals, and appropriate paired statistical tests.

Dataset validity. The quality and representativeness of the machine learning models depend on the characteristics of the training data. Additional experiments using independent benchmark datasets and real-world CI/CD test logs are required to establish generalization.

Reproducibility. The study would benefit from releasing the source code, dataset or dataset-generation procedure, model configurations, experimental scripts, and environment specifications so that the reported results can be independently reproduced.

CONCLUSION

This study presented an AI-powered orchestration pipeline that integrates environment quality prediction and test case prioritization into a unified framework. By leveraging Decision Tree and Random Forest models, the system achieved measurable improvements in early fault detection, defect detection rate, execution efficiency, and test reduction compared to FIFO and heuristic baselines. The findings demonstrate that machine learning can effectively augment orchestration pipelines, reducing human intervention and enhancing CI/CD workflows. While the prototype was evaluated in a controlled environment, the results provide a strong foundation for future work. Subsequent research should focus on scaling the pipeline to cloud-native and multi-tenant infrastructures, incorporating persistent storage and containerized microservices, and validating performance across diverse real-world datasets. By addressing these directions, AI-assisted orchestration can evolve into a robust solution for modern DevOps practices, combining efficiency, adaptability, and sustainability.

Future research should focus on extending the AI-assisted orchestration pipeline beyond the prototype stage into production-scale CI/CD environments. This includes integrating persistent storage and containerized microservices (e.g., Docker and Kubernetes) to ensure scalability and portability, as well as connecting with established CI/CD platforms such as Jenkins, GitHub Actions, or GitLab CI for real-time orchestration. Broader validation using diverse benchmark datasets and real-world test execution logs will be necessary to establish generalization across domains. Additionally, exploring sustainability-aware orchestration by optimizing energy consumption and resource allocation, along with incorporating advanced ensemble or reinforcement learning models, can further enhance adaptability, efficiency, and fault tolerance in multi-tenant and cloud-native infrastructures.

REFERENCES

1. **Adegunwa, O.**, & Solomon, O. S. (2026). A Multi-Layer Security Evaluation Framework for Hardening Kubernetes in Devsecops Pipelines: A Quantitative Experimental Evaluation. *Asian Journal of Research in Computer Science*, *19*(10), 1–18. https://doi.org/10.9734/ajrcos/2026/v19i10913

2. Adekunle, O., Chinwe, E., & Bello, R. (2025). Multilingual transformer models for adaptive orchestration in cloud testing. IEEE Transactions on Software Engineering, 51(2), 210–223.
3. Ajilore, O. O., Eludire, A., Olumoye, M. Y., Adegunwa, O., Omotayo, O., Adekunle, A., ... & Olumoye, Y. (2023). Image classification algorithms for early detection of learning disabilities using visual data. International Journal of Innovative Science and Research Technology, 8(11), 815-819.
4. Ajilore, O. O., Phillip, A., Olumoye, M.Y., Eludire, A., Akanni, A., Adegunwa, O., (2025). Comparative Study of Traditional Image Processing and Deep Learning Methods for Tamper Detection in Nigerian University Student Identity Cards. International Journal of Computer Science and Information Security 19 (4), 109-126
5. Ghosh, S., Chowdhury, P., & Buckchash, H. (2025). AI-driven orchestration for continuous testing pipelines. Journal of Intelligent Systems, 34(4), 112–128.
6. Govindan, V. (2023). AI-powered optimization of non-production environments: Turning constraints into business value. International Journal of Research Publications in Engineering, Technology and Management (IJRPETM), 6(1), 8089-8104.
7. Gopinathan, V. R. (2025). AI-powered Kubernetes orchestration for complex cloud-native workloads. International Journal of Research Publications in Engineering, Technology and Management (IJRPETM), 8(6), 13215-13225.
8. Katta, T. B. (2023). Adaptive AI-driven integration pipelines for efficient data and process orchestration in cloud-native environments. International Journal of Research and Applied Innovations, 6(1), 8363-8374.
9. Kaur, P., & Singh, R. (2024). Dynamic test environment management using machine learning. Software Quality Journal, 32(1), 55–72.
10. Li, H., & Chen, Y. (2024). Predictive resource allocation in automated testing environments. International Journal of Cloud Computing, 18(3), 145–162.
11. Mensah, J., & Okonkwo, C. (2025). Sustainable AI orchestration for distributed software testing. Applied Artificial Intelligence Review, 9(1), 88–104.
12. Pushparathi, V. G. (2024). AI-Powered Enterprise Quality Engineering through Intelligent Test Automation and Secure Cloud Infrastructure Platforms. International Journal of Advanced Research in Computer Science & Technology (IJARCST), 7(5), 11018-11029.
13. Rahman, M., Patel, S., & Osei, K. (2025). DevOps automation through intelligent orchestration frameworks. Journal of Systems and Software, 195, 111–129.
14. Sugumar, R. (2024). AI-Augmented Quality Engineering for Performance Optimization and Test Orchestration in Distributed Systems. International Journal of Science, Research and Technology, 7(5), 12835-12846.
15. Tariq, M. U. (2026). Autonomous Data Orchestration With Generative AI: Redefining Pipelines for Intelligent Analytics. In Generative AI-Powered Data Architectures: From Governance to Autonomous Analytics (pp. 165-192). IGI Global Scientific Publishing.
16. Urtasun, R. (2025). AI Powered Cloud Native Platforms for Intelligent DevOps and Real Time Enterprise Data Systems. International Journal of Research and Applied Innovations, 8(6), 13110-13118.
17. Zhang, L., Chen, Y., & Li, H. (2023). Challenges in test environment orchestration for microservice architectures. Expert Systems with Applications, 224, 119875.