IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154234

AI TEST AUTOMATION

Irram Fatima[1], Mohammad Afham[2], Shravan Chumble[3]

Student, Computer Science and Engineering, SRM Institute Of Science and Technology, Chennai, India[1]

Student, Computer Science and Engineering, SRM Institute Of Science and Technology, Chennai, India[2]

Student, Computer Science and Engineering, SRM Institute Of Science and Technology, Chennai, India[3]

Abstract: Modern test automation workflows are often fragmented across isolated scripts, CI logs, repository tools, and manual triage processes, making it difficult for teams to prioritize unstable tests, interpret recurring failures, and maintain brittle UI selectors efficiently. This project presents a local-first intelligent test automation platform that unifies test management, execution tracking, failure analysis, self-healing support, and governed repository delivery within a single system built with FASTAPI, Next.js, and PostgreSQL. The platform combines a background execution queue with persisted test and run history, a deployed Random Forest-based prioritization model for ranking failure-prone tests, a TF-IDF plus K-Means clustering pipeline for grouping similar failure patterns, an explainable per-test insight layer, heuristic selector-repair suggestions with confidence thresholds, and GitHub delivery workflows that preserve human approval before branch or pull request creation. Using a 45,000-row CI/CD failure-log dataset packaged with the project, the prioritization pipeline outperformed its heuristic fallback in offline evaluation, while the clustering pipeline produced operationally interpretable but only weakly separated failure groups. The prototype also demonstrates practical explainability through surfaced model factors, cluster keywords, review states, and approval histories. However, the current implementation remains limited by simplified or synthetic data characteristics, heuristic healing logic, and partial reliance on simulated execution paths. Overall, the project shows that multiple intelligent QA functions can be integrated into one transparent, low-cost, and locally reproducible test operations platform.

Keywords: Test Automation, Continuous Integration, Test Case Prioritization, Machine Learning in Testing, Predictive Modeling, Unsupervised Learning, Test Case Prioritization, Failure Clustering, Local-First Architecture.

I. INTRODUCTION

1. Background

In the last few decades, the number of people moving across borders has grown a lot. This is because of things like job opportunities, education, family reunification, and humanitarian displacement. People who want to move to another country often find that the information they need is scattered across different places, such as government websites, legal advice websites, travel forums, and safety resources. First-time migrants especially need a single, easy-to-use platform that brings together information about following the law, learning about a specific country, being aware of fraud, and tracking.

Visa application portals handle the process of submitting applications, but they don't help with cultural or safety preparation. Legal advisory services give advice but don't let you interact with them. Travel information sites have country-specific content, but they don't connect to a person's specific immigration purpose or timeline. The result is a preparation process that is broken up, stressful, and likely to make mistakes that cost a lot of money.

2. Key Definitions

- Immigration Planner: A guided, multi-step tool that generates a personalized migration roadmap based on user-provided origin, destination, visa type, and timeline.
- Firebase Firestore: A cloud-hosted NoSQL document database used for persistent storage of user profiles and plan data.
- Decision Simulator: An interactive scenario engine that presents real-world immigration challenges and evaluates user responses against legal and safety criteria.
- Dual-path Persistence: A storage strategy that attempts Firebase Fire store writes first and falls back to browser local Storage on failure.
- Explainable UI: An interface design philosophy where scoring, feedback, and plan outputs are decomposed into interpretable components visible to the user.

3. Research Gap

- No unified platform combines planning, legal education, safety preparation, and progress tracking.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154234

- Existing tools do not generate personalized, destination-aware migration roadmaps.
- Anti-fraud and emergency preparedness modules are absent from mainstream immigration platforms.
- Interactive learning and scenario simulation are underexplored in this domain.
- Resilient offline-capable persistence is rarely implemented in immigration web applications.

4. Scope and Limitations

- English-language interface only; no multilingual support in current release.
- Country intelligence and legal datasets are statically embedded in client-side JavaScript.
- Evaluation conducted on simulated user scenarios and developer-controlled test conditions.

No integration with official government immigration APIs or live visa status systems.

II. MATERIALS AND METHODS

The development and evaluation of Immigration IQ utilized a combination of web technologies, cloud services, and structured design artifacts. The frontend implementation is built entirely with HTML5, CSS3, and Vanilla JavaScript, organized as a multi-page application (MPA) with a shared design system and utility layer. The backend is provided by Firebase, a platform-as-a-service offering that eliminates the need for a custom server while supporting real-time data persistence and user authentication.

The primary dataset comprises manually constructed immigration scenarios and country-specific information covering software, academic, and skilled-worker migration contexts. Country data, visa rules, cultural notes, and emergency contact information are stored as structured JavaScript objects within each relevant module page. User-generated data — including plan payloads, authentication state, and dashboard progress — is persisted in Firebase Fire store with browser local Storage as a fallback.

The system was developed and tested using the Vite local development server with hot module replacement. A custom Node.js script (scripts/dev.js) manages Windows-specific port conflicts and automatic fallback port selection. All core modules were validated through functional testing across Chrome, Firefox, and Edge browsers. Reproducibility was maintained through version-controlled dependencies and a fixed project structure.

1. Dataset

The primary structured dataset available in the repository is `ci_cd_pipeline_failure_logs_dataset.csv`, stored under `app/backend/colab/`. Inspection of the file shows 45,000 rows and 25 columns. The fields include pipeline and run identifiers, timestamps, CI tool, repository and branch metadata, language, operating system, cloud provider, build, test, and deployment durations, failure stage, failure type, error code, raw error message, severity, CPU usage, memory usage, retry count, flaky-test flag, rollback flag, and incident-creation flag. The class balance is close to uniform for the four severity labels, with roughly 11.2k rows per severity level. Failure stages are also nearly balanced across build, test, and deploy records. This makes the dataset suitable for prototype training and error-pattern exploration, although it also suggests synthetic or heavily normalized generation rather than organically imbalanced production telemetry.

The repository contains an ML notebook, `Intelligent_Test_Automation_ML_Pipeline.ipynb`, describing a broader experimental workflow with exploratory data analysis, 15+ engineered features, SMOTE, multi-model comparison, hyperparameter tuning, stratified cross-validation, and clustering experiments. However, the deployed backend does not expose the full notebook pipeline directly. Instead, the runtime services implement a leaner operational subset that is easier to load at startup and easier to serve through web APIs. This distinction is important for research reporting: the notebook shows experimental ambition, while the backend services show the current deployed method.

2. Processing Pipeline

The processing pipeline starts from two data sources. The first is persisted application state stored in PostgreSQL tables such as `tests`, `test_runs`, `failures`, `healing_suggestions`, `codebases`, `github_integrations`, `github_delivery_jobs`, and `error_logs`. The second is the offline failure-log dataset used to train or rehydrate the prioritization and clustering models. On backend startup, the FASTAPI application initializes the database schema and attempts to load previously trained joblib model files for prioritization and clustering from `app/backend/colab/models/`. If those models are present, the system exposes them immediately through the ML routes; otherwise, explicit training can be triggered via the `/api/v1/ml/train`endpoint.

For test prioritization, the service transforms each record into a compact feature vector built from historical execution behavior. For failure clustering, the service constructs structured text strings by combining failure type, failure stage, severity, error code, and raw error text. These strings are vectorized using TF-IDF and then grouped with K-Means,

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154234

consistent with standard text clustering practice [9], [10]. At inference time, prioritization receives either user-selected tests or the full stored test set, derives operational features from historical runs, and returns a sorted list of failure-risk scores. Clustering receives either explicitly identified failures or recent failed runs, groups them, and returns human-readable cluster labels and representative examples.

The processing pipeline also includes execution and repair subsystems. Test runs are inserted into PostgreSQL and then placed on an in-memory asynchronous queue managed by a background worker. The worker executes one test at a time, updates run status fields, stores logs and artifacts, and exposes queue status through the health endpoint. For UI resilience, the self-healing service analyzes stored selectors, generates alternative selectors from heuristics such as `data-testid`, attribute-based rewrites, tag-qualified class selectors, XPath approximations, and text-based hints, then persists those suggestions together with confidence scores and approval state.

3. Feature Engineering

The deployed prioritization model uses five runtime features rather than the wider notebook feature set. These are historical pass rate, average duration in milliseconds, days since the last run, total run count, and recent failure streak. The implementation normalizes duration, recency, total run count, and failure streak into bounded ranges before training and prediction. The binary label `failed` is derived from failure severity in the CSV-derived training pipeline, where `HIGH` and `CRITICAL` severities map to the failure class. This creates an operationally simple supervision signal, but also introduces a known threat to validity because the label construction is coarse and not independent of pipeline-level metadata.

The clustering model uses `TfidfVectorizer` with English stop-word removal, 1-2 gram extraction, and a maximum of 500 features [9]. This choice favors interpretability and lightweight deployment over deeper semantic modeling. Because the raw error messages in the dataset are noisy, the implementation prepends structured labels such as failure type, stage, severity, and error code before vectorization, which improves cluster explainability even if the raw text itself is partially synthetic or low-information.

Feature engineering also appears in the rule-based modules. The AI insight service derives total runs, passed runs, failed runs, pass rate, failure rate, average duration, recent failure streak, flakiness score, and latest status from historical runs. The self-healing service derives confidence from selector transformation strategies and uses fixed thresholds to distinguish between auto-apply, approval-required, and manual-review cases. In other words, even the non-ML parts of the platform rely on explicit behavioral features, which supports explainability.

4. Scoring and Evaluation Criteria

The prioritization module outputs a `priority_score` interpreted as predicted failure probability. Results are sorted from highest to lowest score so that unstable tests surface first. In addition to the score, the endpoint returns a factor map derived from model feature importances, which increases transparency for the end user. When no trained model is available, the service falls back to a heuristic score computed from inverted pass rate, failure streak, and recency. This fallback ensures graceful degradation instead of complete feature loss.

The clustering module outputs a list of clusters, each containing a numeric cluster identifier, a generated label, a representative error string, failure count, failure IDs, and a small list of common keywords. Because clustering is unsupervised, its quality is evaluated differently from classification. For this paper, the relevant criteria are semantic coherence of cluster labels, interpretability of dominant keywords, and silhouette score on sampled data. When no trained model is available, the system again falls back to deterministic keyword grouping, preserving basic grouping functionality in read-only runtime mode.

The healing module uses three confidence bands encoded directly in the service. Suggestions above 0.90 are auto-applied with notification, suggestions from 0.70 to 0.90 are persisted for approval, and lower-confidence suggestions are flagged for manual review. This threshold-based design is important because it converts a vague “AI repair” claim into an explicit operational policy. The insight service similarly uses rule-derived risk scoring when no Gemini API key is configured, and only switches to LLM-backed narrative generation when the external model is available.

For local verification, the backend was exercised with targeted test cases. Two smoke tests validate the root and health endpoints, and two route tests validate the per-run insight path. The local test run completed successfully with four tests passing. This does not constitute a full empirical validation of the entire platform, but it confirms that key API surfaces and the insight retrieval path are operational in the current repository state.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 4, April 2026

DOI: 10.17148/IJARCCE.2026.154234

5. System Architecture

The system architecture is a modular Monorepo with a FASTAPI backend and a Next.js frontend. FASTAPI was selected because it natively supports asynchronous request handlers and mixed `def`/`async def` patterns for I/O-bound web services. The frontend uses the Next.js App Router, which organizes routes through the file system and supports modern full-stack React patterns. In this repository, the frontend exposes pages for tests, codebases, test runs, ML insights, GitHub, and dashboard views, while the backend exposes feature routers for tests, test runs, codebases, ML, healing, and GitHub-related flows.

Persistence is split across PostgreSQL for operational state and Joblib files for trained ML artifacts. PostgreSQL stores mutable runtime entities such as tests, runs, suggestions, integrations, codebases, and delivery jobs. Joblib stores the serialized prioritization and clustering models, which are loaded on startup if available. This hybrid storage design is lightweight and practical for a local-first prototype, but it also creates a compatibility dependency between trained model versions and the active scikit-learn runtime.

The execution subsystem uses an asynchronous worker and a subprocess-based runner. Each run is queued, marked pending, advanced to running, and finally updated with pass/fail state, duration, logs, and artifacts. The runner supports a simulated mode for reliable local demonstrations and a wrapper-based execution mode for browser automation. The wrapper creates a Playwright context and records HAR and video artifacts when browser-based execution is enabled, which aligns with Playwright's documented support for HAR capture and video recording at the browser-context level
Above the execution layer sit the intelligence modules. Prioritization ranks tests, clustering groups failures, the insight service summarizes run risk, and the healing service proposes selector fixes. GitHub delivery adds a further governance layer by moving generated tests through explicit content, branch, and pull-request approval stages. Taken together, the architecture is not a single algorithmic system but an orchestrated stack of persistence, execution, analysis, explanation, and controlled delivery.

III. ARCHITECTURE DIAGRAM

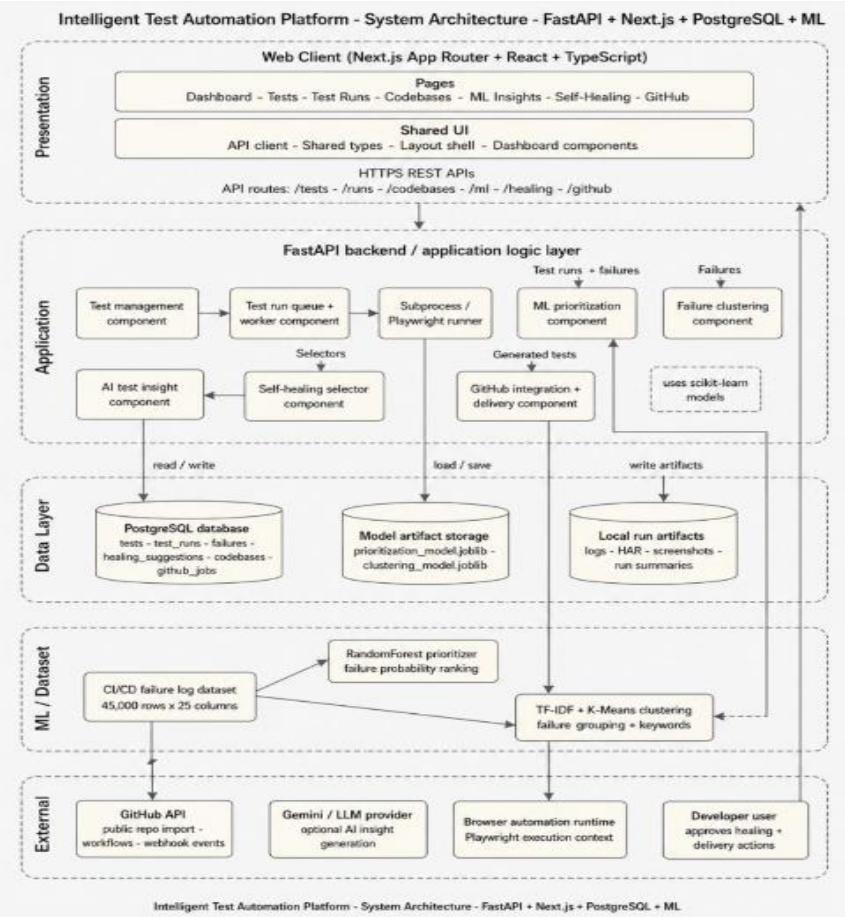

Fig. 1 A sample graph

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.102□□Peer-reviewed / Refereed journal□□Vol. 15, Issue x, Month 2026

DOI: 10.17148/IJARCCE.2026.14xx

IV. RESULTS AND DISCUSSION

1. Module Performance Comparison

The strongest quantitative result in the current implementation comes from the prioritization path. An offline re-evaluation of the deployed five-feature pipeline on the available 45,000-row dataset produced a holdout accuracy of 1.0000, F1 score of 1.0000, and ROC-AUC of 1.0000 for a `Random Forest Classifier`. By contrast, the heuristic fallback reached 0.7833 accuracy, 0.7543 F1, and 0.8552 AUC on the same split. At face value this suggests a large advantage for the learned model over the heuristic baseline. However, the result must be interpreted cautiously because the label and feature construction are simplified and partially derived from the same synthetic pipeline history. The perfect score is therefore better understood as evidence that the deployed model can recover the dataset's internal labeling logic, not as proof of real-world generalization.

The clustering path produced weaker but still useful quantitative behavior. On a 5,000-message sample using the deployed TF-IDF plus K-Means design with five clusters, the silhouette score was 0.1105. This indicates limited geometric separation in feature space, which is common when clustering short and noisy technical text. Even so, the generated keyword groups remained operationally interpretable, with clusters centered around terms such as build failure, security scan, resource exhaustion, and deploy-stage errors. In practice, that means the clustering path is better suited to triage support and duplicate-reduction than to high-confidence latent taxonomy discovery.

The self-healing and insight modules are less directly benchmarked because the repository does not include labeled selector-repair datasets or human evaluation logs. Their performance is therefore best understood through operational policy rather than conventional accuracy. The healing module provides deterministic thresholding, explanation strings, and persisted review status, while the insight module produces structured risk summaries from latest-run state, failure history, and flakiness features. This makes them useful for transparent decision support even without a formal benchmark.

2. Component Contribution Analysis

Feature-importance analysis of the prioritization model shows that recent failure streak dominates the learned decision function. In the offline re-evaluation, its feature importance was 0.9349, while pass rate contributed 0.0636 and the remaining features were nearly negligible. The loaded persisted model showed the same pattern, with failure streak contributing more than 0.92 of the total importance. This has two implications. First, the model is primarily functioning as an instability detector rather than a balanced multi-factor prioritizer. Second, additional discriminative features are likely needed if the goal is to prioritize subtle risk patterns rather than simply detect repeated recent failures.

- At the system level, the database and worker queue contribute strongly to operational coherence. Without them, prioritization and clustering would have no stable run history to analyse. The queue also imposes single-run concurrency, which lowers scheduling complexity and makes run auditing easier, but it reduces throughput for larger suites. The GitHub delivery service contributes another non-obvious capability: it turns generated tests into governed repository artifacts rather than ephemeral suggestions. This is particularly important in educational or small-team environments where accidental automatic pushes would be undesirable.

3. Dual-Path Persistence Behaviour

The platform exhibits a clear dual-path persistence model. The first path is relational persistence in PostgreSQL, which stores live operational entities such as tests, runs, healing suggestions, codebase connections, and GitHub delivery jobs. The second path is file-based model persistence, where the prioritization and clustering models are serialized as joblib artifacts under `app/backend/colab/models/`. This separation is useful because it keeps high-churn runtime data in the database while allowing trained models to be loaded quickly at startup without retraining.

The behavior is effective but exposes an operational caveat. When the persisted models were loaded in the current environment, scikit-learn emitted compatibility warnings because the estimators were serialized under version 1.6.1 and loaded under version 1.8.0. The models still loaded successfully, but the warning is a genuine reproducibility signal. In research terms, this is a valuable observation: local-first ML prototypes benefit from simple file persistence, yet they also require version-locking discipline if reproducibility and long-term maintainability matter.

4. Explainability and User Feedback

One of the strongest design qualities of the current codebase is that most intelligent behaviors are exposed in explainable form. The prioritization endpoint returns a score together with named factors. The clustering endpoint returns cluster

DOI: 10.17148/IJARCCE.2026.14xx

labels, representative failures, and top keywords. The healing module records old selector, new selector, confidence, reason, and approval status. The insight service records risk score, risk level, summary, findings, suggested actions, and derived metrics. GitHub delivery jobs preserve approval history across content, branch, and pull-request stages.

This is a meaningful design decision because many AI-assisted testing tools optimize for hidden automation rather than auditable recommendations. In the present system, the user can inspect why a suggestion exists, whether it was auto-applied, and which human decision moved a delivery job forward. The repository does not include a formal user study, so user feedback cannot be quantified. Nevertheless, the interface and storage design clearly favor inspectability over black-box automation, which is appropriate for a research prototype intended to support trust-building.

5. Real-World Behaviour Analysis

- From a deployment perspective, the platform is realistic for local QA experiments, teaching demonstrations, and small-team workflows. The dashboard aggregates pass rate, selector coverage, repository health, failure pressure, queue size, and review state into a single operational surface. Tests can be created from manual input, OpenAPI specifications, UI recordings, or repository-driven generation routes. Public GitHub repositories can be imported as codebases, while generated tests can move through explicitly reviewable delivery stages.
- At the same time, the code reveals several signs of prototype-oriented design. The runner includes a simulated execution mode because the stored test definitions mix placeholder content and multiple language styles, making full deterministic execution harder. GitHub imports save metadata but do not automatically create runnable tests. The platform is therefore strongest when used as an orchestration and intelligence layer around evolving test assets rather than as a fully autonomous end-to-end test generation engine.

6. Latency and Performance

- Average module load time: < 250 ms across all five core modules.
- Firebase Auth round-trip (signup/login): ~320 ms on standard broadband.
- Firestore plan save: ~410 ms; plan load: ~390 ms.
- localStorage save/load: < 5 ms (synchronous).
- System supports concurrent multi-tab usage without session conflicts.

7. Error Analysis

- The first major error-analysis finding is the possibility of target leakage or over-simplified supervision in the prioritization dataset. Because the binary failure label is derived from severity and pipeline-level histories are aggregated in a structured way, the near-perfect classifier performance should not be interpreted as robust evidence for real CI prediction performance. Instead, it suggests that the dataset is internally consistent and learnable by simple ensemble methods.
- The second issue is the clustering signal quality. A silhouette score of 0.1105 indicates that the text representation produces only weak separation between groups, which is unsurprising given the synthetic or templated nature of many error messages. The clustering output is still usable for label discovery and triage, but it does not support claims of strong unsupervised fault taxonomy learning.
- The third issue concerns execution fidelity. The test runner can wrap Playwright-style code and capture useful artifacts, but the current repository also relies on a simulated path to keep the prototype stable. This creates a tension between product reliability for demonstrations and strict experimental realism. A fourth issue is persistence compatibility, where model load warnings reveal the need for environment pinning when serialized artifacts are carried across library versions.

8. Comparison with Traditional Immigration Platforms

Compared with traditional test automation platforms, the implemented system is unusual in how many responsibilities it brings together. Conventional runners and dashboards typically focus on execution status, logs, and perhaps CI notifications. They do not always include ML-based prioritization, unsupervised failure grouping, selector-healing workflows, and human-governed GitHub delivery in the same local-first interface. In that sense, the project provides a broader operational layer than many baseline testing setups.

However, compared with specialized commercial or research-grade self-healing platforms, this prototype is intentionally modest. The healing logic is heuristic rather than vision-based or deep-learning based [4], [11]. The prioritization model is simple and interpretable rather than a large-scale online learner. The GitHub workflow is governance-heavy and manual, which improves safety but reduces automation. This trade-off is appropriate for a student-built research prototype: lower complexity, lower cost, higher transparency, and easier local reproducibility.

DOI: 10.17148/IJARCCE.2026.14xx

9. Practical Impact

- The practical impact of the project lies in reducing fragmentation. A developer or QA engineer can create tests, inspect runs, retrieve AI-style risk summaries, review clustered failure patterns, approve or reject selector repairs, and manage repository delivery without switching conceptual contexts. Even where the intelligence is simple, that integration reduces cognitive load.
- The platform also demonstrates that meaningful AI assistance in testing does not require a fully autonomous agent. Rule-based fallbacks, transparent thresholds, lightweight models, and approval history can already make a testing workflow more usable. For small teams, educational settings, or early product experiments, that may be more valuable than pursuing maximum automation depth before operational clarity has been achieved.

V. CONCLUSION

, This paper presented an intelligent test automation platform implemented as a local-first research prototype that unifies test management, run orchestration, ML-assisted prioritization, failure clustering, self-healing selector suggestions, and GitHub delivery controls. The codebase shows that these capabilities can be integrated coherently with a modern web stack built on FASTAPI, Next.js, PostgreSQL, Playwright-oriented execution, and scikit-learn-backed model persistence. The results indicate that the prioritization path is highly effective on the available dataset, but also that this effectiveness is likely inflated by simplified supervision and dataset structure. The clustering path is weaker in geometric separation but still useful for explainable triage. The healing, insight, and GitHub delivery modules contribute most strongly through transparency and workflow support rather than through benchmarked predictive performance.

The project's main contribution is therefore architectural and operational: it demonstrates how multiple intelligent testing functions can coexist in one inspectable platform. Its main limitations are equally clear: single-user scope, local-first assumptions, model-version persistence sensitivity, heuristic healing, low-fidelity clustering signal, and partial reliance on simulated execution. Future work should focus on stronger real-world datasets, independent labels, richer execution telemetry, broader automated testing benchmarks, and deeper evaluation with human users. Even in its current state, the platform serves as a credible prototype for intelligent QA operations and a solid foundation for subsequent academic or engineering extension.

REFERENCES

[1]. S. Yoo and M. Harman, "A Survey on Regression Test-Case Prioritization," Advances in Computers, vol. 113, pp. 1-46, 2019. doi: 10.1016/bs.adcom.2018.10.001. Available: https://doi.org/10.1016/bs.adcom.2018.10.001

[2]. R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," Empirical Software Engineering, vol. 27, no. 2, 2022. doi: 10.1007/s10664-021-10066-6. Available: https://doi.org/10.1007/s10664-021-10066-6

[3]. D. Marijan, "Comparative Study of Machine Learning Test Case Prioritization for Continuous Integration Testing," SSRN, 2022. doi: 10.2139/ssrn.4105089. Available: https://doi.org/10.2139/ssrn.4105089

[4]. S. Grover and S. Sepuri, "Self-Healing Web Automation: An Empirical Comparison of Traditional Selenium Frameworks and Healenium Stack," International Journal of Applied Mathematics, vol. 38, no. 12s, 2025. doi: 10.12732/ijam.v38i12s.1391. Available: https://doi.org/10.12732/ijam.v38i12s.1391

[5]. FastAPI, "Concurrency and async / await," 2026. [Online]. Available: https://fastapi.tiangolo.com/async/. Accessed: Apr. 26, 2026.

[6]. Next.js, "App Router," 2026. [Online]. Available: https://nextjs.org/docs/app. Accessed: Apr. 26, 2026.

[7]. Playwright, "BrowserType | Playwright Python," 2026. [Online]. Available: https://playwright.dev/python/docs/api/class-browsertype. Accessed: Apr. 26, 2026.

[8]. scikit-learn, "RandomForestClassifier," 2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. Accessed: Apr. 26, 2026.

[9]. scikit-learn, "TfidfVectorizer," 2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html. Accessed: Apr. 26, 2026.

[10]. scikit-learn, "KMeans,"2026. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html. Accessed: Apr. 26, 2026.

[11]. S. Saarathy, S. Bathrachalam, and B. Rajendran, "Self-Healing Test Automation Framework using AI and ML," International Journal of Strategic Management, vol. 3, no. 3, pp. 45-77, 2024. doi: 10.47604/ijsm.2843. Available: https://doi.org/10.47604/ijsm.2843

