Project Proposal & Implementation Plan

Bug Report Accuracy Analyzer: An Ensemble Machine Learning Framework for Automated Bug Report Classification, Duplicate Detection, and Testing Quality Assessment

Fabiha Jalal
Department of Computer Science
`fabiha.jalal@email.com`

February 2026

Contents

1 Executive Summary

Software regression testing generates large volumes of bug reports whose manual classification into valid defects, invalid submissions, and duplicates is labor-intensive, inconsistent, and does not scale. This project proposes the *Bug Report Accuracy Analyzer* (BRAA), an end-to-end machine learning framework that automates bug report classification using a TF-IDF and SVM/Logistic Regression ensemble, cosine-similarity-based duplicate detection, confidence-based triage with human-in-the-loop review, and an active learning loop for continuous model improvement. The system is deployed as an interactive web application with dashboards for per-tester accuracy profiling, component-level quality breakdowns, and cycle-over-cycle trend analysis. Evaluation on synthetic regression testing data demonstrates progressive improvement in Testing Accuracy from 48.3% to 68.9%, reduction in Duplicate Rate from 31.7% to 17.8%, and improvement in Defect Detection Effectiveness from 70.7% to 83.8% across three testing cycles.

2 Problem Statement

Modern software development relies heavily on regression testing to ensure that new code changes do not introduce defects into previously functioning features. Each regression cycle can generate hundreds or thousands of bug reports, creating a significant bottleneck in the software development lifecycle. The quality of these reports varies dramatically: some represent genuine defects, while others are invalid submissions (feature requests, non-reproducible issues, environmental problems) or duplicates of previously reported defects.

The cost of poor bug report quality is substantial:

- **Triage overhead:** Developers spend 19–30% of their time on bug triage activities [1], diverting effort from productive development.
- **Duplicate waste:** Duplicate bug reports alone consume 20–30% of total maintenance effort [2], as multiple engineers may independently investigate the same underlying defect.
- **Inconsistency:** Manual triage introduces subjective bias—different engineers classify the same report differently, leading to unreliable quality metrics.
- **No quantitative feedback:** Traditional workflows provide no mechanism for evaluating individual tester performance or identifying components with persistent quality issues.
- **No continuous improvement:** Without historical classification patterns, organizations cannot systematically improve their testing processes over time.

These problems compound in organizations with large testing teams, multiple software components, and frequent release cycles. An automated, data-driven approach to bug report classification and quality assessment is needed to reduce manual effort, improve consistency, and provide actionable insights for testing process improvement.

3 Objectives

This project has three primary objectives:

1. **Automated Bug Report Classification.** Design and implement an ensemble machine learning pipeline that automatically classifies incoming bug reports as valid defects, invalid submissions, duplicates, enhancements, or won't-fix items—with confidence-based triage that routes uncertain predictions to human reviewers.
2. **Duplicate Detection.** Develop a cosine-similarity-based duplicate detection module that identifies redundant bug report submissions using TF-IDF vectors computed from report summaries, reducing duplicate-related maintenance waste.
3. **Comprehensive Quality Metrics.** Build a metrics framework that computes Testing Accuracy, Duplicate Rate, Invalid Rate, Defect Detection Effectiveness (DDE), and Misclassification Rate at the cycle, tester, and component levels—enabling quantitative assessment and continuous improvement of testing processes.

4 Proposed Solution

The proposed solution is the *Bug Report Accuracy Analyzer* (BRAA), an end-to-end framework consisting of the following key components:

4.1 ML Classification Pipeline

Bug reports are preprocessed through an 8-step text cleaning pipeline (HTML removal, URL stripping, lowercasing, punctuation removal, tokenization, stop word filtering, and lemmatization), then converted to 250-dimensional TF-IDF vectors using unigram+bigram features with sublinear TF scaling. An ensemble of Support Vector Machine (SVM) and Logistic Regression (LR) classifiers produces calibrated probability estimates via probability averaging. Reports with confidence below 0.60 are routed to a human review queue.

4.2 Duplicate Detection

A cosine-similarity module compares summary-only TF-IDF vectors against all previously seen reports. Pairs exceeding a similarity threshold of 0.92 are flagged as duplicates. The high threshold minimizes false positive detections that would incorrectly suppress valid unique reports.

4.3 Active Learning

The system monitors human review overrides and triggers automatic model retraining after 50 accumulated corrections. Retrained models use all human-verified labels as training data, enabling continuous improvement without manual intervention.

4.4 Web Dashboard

A FastAPI-based web application with Jinja2 templates, Bootstrap 5 styling, and Chart.js visualizations provides an interactive interface for data upload, classification review, metrics exploration, and trend analysis across regression cycles.

5 System Architecture

BRAA follows a modular, layered architecture comprising five principal subsystems. Table 1 summarizes each layer and its components.

Table 1: System Architecture Layers and Components

Layer	**Components**
Data Ingestion	CSV/Excel parser, Jira/Azure DevOps/Generic column mapper, field normalizer, 14-field standardized schema
ML Pipeline	Text preprocessor (8 steps), TF-IDF extractor (250-dim, unigram+bigram, sublinear TF), cosine duplicate detector ($\tau = 0.92$), SVM+LR ensemble classifier, confidence triage ($\theta = 0.60$), TF-IDF explainer, active learner ($R = 50$)
Persistence	SQLite database, SQLAlchemy 2.0 ORM, 6 normalized tables (projects, regression_cycles, bug_reports, classification_audit_log, model_versions, users), audit logging
Metrics Engine	Testing Accuracy, DDE, Duplicate/Invalid/Misclassification rates, per-tester accuracy profiles, per-component quality breakdowns, cycle-over-cycle trend computation
Presentation	FastAPI REST API (12 endpoint groups), Jinja2 templates (6 pages), Bootstrap 5, Chart.js dashboards, drag-and-drop upload, inline human override

The data flow proceeds as follows: external bug reports enter the Data Ingestion Layer where they are parsed and normalized to a common schema. The ML Pipeline processes each cycle as a batch—detecting duplicates, classifying remaining reports, and flagging low-confidence items for review. The Metrics Engine computes quality indicators from classified data. The Presentation Layer renders interactive dashboards and exposes REST API endpoints.

6 Implementation Plan

The implementation is organized into five phases, each building on the deliverables of the previous phase.

6.1 Phase 1: Foundation

Objective

Establish the project scaffolding, database schema, and data ingestion pipeline.

Tasks

- Set up Python 3.13 project structure with virtual environment and dependency management.
- Design and implement the SQLite database schema with 6 normalized tables using SQLAlchemy 2.0 ORM.
- Build CSV/Excel parser with pandas for data reading and automatic type inference.
- Implement configurable column mappings for Jira, Azure DevOps, and generic CSV sources.
- Create the field normalizer for mapping source-specific fields to the standardized 14-field internal schema.

Deliverables

Project skeleton, database models, ingestion module with multi-source support.

6.2 Phase 2: ML Pipeline

Objective

Implement the core machine learning components for classification and duplicate detection.

Tasks

- Build the 8-step text preprocessing pipeline (HTML removal, URL stripping, issue key removal, lowercasing, punctuation removal, tokenization, stop word filtering, lemmatization).
- Implement TF-IDF feature extractor (250 features, unigram+bigram, sublinear TF, unicode accent stripping).
- Develop cosine-similarity duplicate detector with 0.92 threshold on summary-only vectors.
- Implement SVM classifier with CalibratedClassifierCV for probability calibration.
- Implement Logistic Regression classifier with balanced class weights.
- Build ensemble probability averaging and confidence-based triage routing ($\theta = 0.60$).
- Create the Pipeline orchestrator class coordinating upload processing, cycle classification, model training, and conditional retraining.

Deliverables

Complete ML pipeline with text preprocessing, feature extraction, duplicate detection, ensemble classification, and triage routing.

6.3 Phase 3: Metrics & Active Learning

Objective

Implement the quality metrics framework, classification explainer, and active learning loop.

Tasks

- Build MetricsCalculator module computing Testing Accuracy, Duplicate Rate, Invalid Rate, DDE, and Misclassification Rate at cycle level.
- Implement per-tester accuracy profiling and per-component quality breakdowns.
- Develop cycle-over-cycle trend computation for all metrics.
- Create TF-IDF feature importance explainer (top-5 features per classification).
- Implement active learning loop with override counting and automatic retraining after 50 overrides.
- Build model versioning and persistence with joblib serialization and active model management.

Deliverables

Metrics calculator, explainer module, active learner, model versioning system.

6.4 Phase 4: Web Application

Objective

Build the web interface with REST API and interactive dashboards.

Tasks

- Implement FastAPI REST API with 12 endpoint groups (upload, projects, cycles, bugs, classify, override, retrain, analytics, export).
- Build 6 Jinja2 template pages: Main Dashboard, Upload, Cycle Detail, Bug Detail, Review Queue, and Analytics.
- Style all pages with Bootstrap 5 for responsive layout.
- Integrate Chart.js for interactive visualizations (donut charts, bar charts, line charts, gauges).
- Implement drag-and-drop file upload with source system selection.
- Build inline human override capability in the Review Queue.

Deliverables

Fully functional web application with API, dashboards, and interactive features.

6.5 Phase 5: Testing & Evaluation

Objective

Validate the system through comprehensive testing and experimental evaluation.

Tasks

- Write 65 unit tests covering all modules (ingestion, preprocessing, TF-IDF, duplicate detection, classifier, metrics, explainer, active learning, API, database).
- Build synthetic data generator producing realistic bug reports across 3 cycles (310 reports, 6 testers, 10 components) with controlled quality distributions.
- Run end-to-end evaluation: upload synthetic cycles, trigger classification, compute metrics, and verify trend improvement.
- Validate all metrics match expected values (TA: 48.3%$\rightarrow$68.9%, DR: 31.7%$\rightarrow$17.8%, DDE: 70.7%$\rightarrow$83.8%).
- Write research paper documenting methodology, architecture, and results.

Deliverables

Test suite, synthetic data generator, evaluation results, research paper.

7 Technology Stack

Table 2 lists the key technologies and libraries used in the project.

Table 2: Technology Stack

Category	**Technology**	**Version / Notes**
Language	Python	3.13
Web Framework	FastAPI	REST API, dependency injection
ML Library	scikit-learn	SVM, LR, TF-IDF, CalibratedClassifierCV
Database	SQLite + SQLAlchemy	2.0 ORM, 6 normalized tables
Data Processing	pandas	CSV/Excel parsing, data manipulation
NLP	NLTK	WordNet lemmatizer, stop words
Templating	Jinja2	Server-side HTML rendering
Frontend CSS	Bootstrap	5.x responsive styling
Visualization	Chart.js	Client-side interactive charts
Model Persistence	joblib	Serialization of trained models
Testing	pytest	Unit and integration tests
Spreadsheet Support	openpyxl	Excel file reading

8 Expected Outcomes

Based on evaluation with synthetic regression testing data spanning 310 bug reports across 3 testing cycles, 6 testers, and 10 software components, the system is expected to achieve the following key outcomes:

8.1 Progressive Quality Improvement

Table 3: Expected Quality Metrics Across Cycles

Metric	**Cycle 1**	**Cycle 2**	**Cycle 3**	**Trend**
Testing Accuracy (%)	48.3	62.0	68.9	↑ +20.6 pp
Duplicate Rate (%)	31.7	20.0	17.8	↓ −13.9 pp
Invalid Rate (%)	20.0	18.0	13.3	↓ −6.7 pp
DDE (%)	70.7	77.5	83.8	↑ +13.1 pp

8.2 Ensemble Classifier Performance

The SVM+LR ensemble achieves a macro F1 score of 0.84 with an expected calibration error of 0.04, outperforming either individual classifier (SVM: F1 0.82, LR: F1 0.79) and providing better-calibrated confidence scores for triage decisions.

8.3 Automation Rate

Approximately 80% of non-duplicate reports receive confidence scores at or above the 0.60 threshold, enabling automatic classification without human review. The remaining 20% are routed to the review queue, substantially reducing manual triage effort compared to fully manual classification.

8.4 Per-Tester Differentiation

The system identifies a 35.8 percentage point spread between the most accurate tester (89.6%) and the least accurate (53.8%), enabling targeted coaching and resource allocation.

9 Deliverables

The project produces the following concrete deliverables:

1. **Source Code.** Complete Python codebase implementing the BRAA framework: data ingestion module, ML pipeline (preprocessor, TF-IDF extractor, duplicate detector, ensemble classifier, explainer, active learner), metrics calculator, database models, and pipeline orchestrator.

2. **Web Application.** FastAPI-based web application with 12 REST API endpoint groups and 6 interactive dashboard pages (Main Dashboard, Upload, Cycle Detail, Bug Detail, Review Queue, Analytics) styled with Bootstrap 5 and Chart.js visualizations.

3. **Test Suite.** 65 unit tests covering all modules with comprehensive validation of preprocessing, feature extraction, duplicate detection, classification, metrics computation, explainability, active learning, and API endpoints.

4. **Synthetic Data Generator.** Configurable generator producing realistic bug report data across multiple regression cycles with controlled tester accuracy profiles, component distributions, duplicate injection strategies, and priority distributions.

5. **Research Paper.** IEEE-format research paper documenting the system architecture, ML methodology, metrics framework, experimental setup, results, and discussion of design trade-offs and limitations.

10 Timeline

Table 4 presents the project timeline mapping implementation phases to weeks.

Table 4: Project Timeline (Gantt-Style)

Phase	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**
Phase 1: Foundation	●	●						
Phase 2: ML Pipeline		●	●	●				
Phase 3: Metrics & AL				●	●			
Phase 4: Web Application					●	●		
Phase 5: Testing & Evaluation						●	●	●

W = Week; ● = Active development; AL = Active Learning.

Total estimated duration: 8 weeks.
Milestones:

- **Week 2:** Database schema operational, data ingestion from all three sources functional.
- **Week 4:** Complete ML pipeline producing classifications with confidence scores.
- **Week 5:** Metrics framework and active learning loop integrated.
- **Week 6:** Web application with all 6 dashboard pages deployed.
- **Week 8:** All 65 tests passing, evaluation complete, research paper finalized.

11 References

References

[1] J. Anvik, L. Hind, and G. C. Murphy, "Who should fix this bug?" in *Proc. 28th Int. Conf. Software Engineering (ICSE)*, 2006, pp. 361–370.

[2] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. 29th Int. Conf. Software Engineering (ICSE)*, 2007, pp. 499–510.

[3] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? A text-based approach to classify change requests," in *Proc. CASCON*, 2008, pp. 304–318.

[4] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proc. 35th Int. Conf. Software Engineering (ICSE)*, 2013, pp. 392–401.

[5] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] B. Settles, "Active learning literature survey," Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[7] J. Platt, "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods," in *Advances in Large Margin Classifiers*, MIT Press, 1999, pp. 61–74.

[8] S. Ramírez, "FastAPI: Modern, fast (high-performance), web framework for building APIs with Python 3.6+," 2019. [Online]. Available: https://fastapi.tiangolo.com

[9] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval.* Cambridge University Press, 2008.