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
Email: fabiha.jalal@email.com

***Abstract*—Software regression testing generates large volumes of bug reports whose quality varies significantly across testers, components, and testing cycles. Manual classification of bug reports into valid defects, invalid submissions, and duplicates is labor-intensive, inconsistent, and does not scale. We present the *Bug Report Accuracy Analyzer* (BRAA), an end-to-end machine learning framework that automates bug report classification using an ensemble of Support Vector Machine (SVM) and Logistic Regression (LR) classifiers operating on TF-IDF feature representations. The system incorporates cosine-similarity-based duplicate detection with a 0.92 similarity threshold, confidence-based triage routing for human review, and an active learning loop that triggers model retraining after accumulating 50 human overrides. We evaluate BRAA on synthetic regression testing data spanning three testing cycles (310 bug reports, 6 testers, 10 components) and demonstrate progressive improvement in testing accuracy from 48.3% to 68.9%, reduction in duplicate rates from 31.7% to 17.8%, and improvement in Defect Detection Effectiveness (DDE) from 70.7% to 83.8% across cycles. The framework provides per-tester accuracy profiling, component-level quality breakdowns, and explainable classifications through TF-IDF feature importance analysis. Our system is deployed as a web application with an interactive dashboard, supporting data ingestion from Jira, Azure DevOps, and generic CSV sources.**

***Index Terms*—bug report classification, duplicate detection, ensemble learning, software quality assurance, TF-IDF, active learning, regression testing, defect detection effectiveness**

I. Introduction

Software regression testing is a critical quality assurance activity that aims to verify that new code changes do not introduce defects into previously functioning features [1]. In large-scale software projects, each regression cycle can generate hundreds or thousands of bug reports, creating a significant bottleneck in the software development lifecycle. The quality of these bug reports varies dramatically: some represent genuine defects requiring immediate attention, while others are invalid submissions (feature requests misclassified as bugs, not-reproducible issues, or environmental problems) or duplicates of previously reported defects [2].

The cost of poor bug report quality is substantial. Studies have shown that developers spend 19–30% of their time on bug triage activities [3], and duplicate bug reports alone can consume 20–30% of total maintenance effort [4]. Furthermore, inconsistent quality across testers makes it difficult for test managers to assess the effectiveness of their testing processes and allocate resources appropriately.

Traditional approaches to bug report management rely heavily on manual triage, where experienced engineers classify incoming reports. This approach suffers from several limitations: (1) it does not scale with increasing project size, (2) it introduces subjective bias and inconsistency, (3) it provides no quantitative metrics for evaluating tester performance, and (4) it offers no mechanism for continuous improvement based on historical classification patterns.

Recent advances in machine learning for software engineering have demonstrated promising results in automating various aspects of bug report management, including classification [5], duplicate detection [8], and severity prediction [6]. However, most existing approaches focus on individual tasks in isolation and lack integration into a cohesive framework that addresses the full lifecycle of bug report quality management.

In this paper, we present the *Bug Report Accuracy Analyzer* (BRAA), an end-to-end framework that combines automated classification, duplicate detection, confidence-based triage, and active learning into a unified system. Our key contributions are:

1) **Integrated ML Pipeline:** We design and implement a complete machine learning pipeline that combines TF-IDF feature extraction (250-dimensional, unigram+bigram, sublinear TF), cosine-similarity duplicate detection (threshold 0.92), and an SVM+LR ensemble classifier with calibrated probability averaging, achieving automated classification with confidence-based human-in-the-loop triage.
2) **Comprehensive Metrics Framework:** We define and implement a multi-dimensional quality metrics framework that computes Testing Accuracy, Duplicate Detection Rate, Invalid Report Rate, Defect Detection Effectiveness (DDE), Misclassification Rate, and per-tester/per-component breakdowns, enabling quantitative assessment of testing quality across cycles.
3) **Active Learning with Explainability:** We implement a confidence-based active learning loop that routes low-confidence predictions (below 0.60) to human reviewers and triggers automatic model retraining after 50 human overrides, combined with TF-IDF feature importance

explanations that provide transparency into classification decisions.

The remainder of this paper is organized as follows. Section II reviews related work in bug report classification, duplicate detection, and machine learning for software engineering. Section III presents the system architecture. Section IV describes the machine learning methodology in detail. Section V defines the metrics framework. Section VI covers implementation details. Section VII describes the experimental setup. Section VIII presents results and evaluation. Section IX discusses design trade-offs and limitations. Section X outlines future directions, and Section XI concludes the paper.

II. Related Work

A. Bug Report Classification

Automated classification of bug reports has been studied extensively in the software engineering literature. Antoniol et al. [5] were among the first to apply text mining techniques to distinguish between bug reports and non-bug issues (such as feature requests), using Naïve Bayes, Logistic Regression, and Decision Trees on three open-source projects. Their work demonstrated that textual features alone could achieve classification accuracy exceeding 80%. Herzig et al. [7] extended this line of research by conducting a large-scale study of misclassified bug reports in five open-source projects, finding that 33.8% of all bug reports were misclassified, with significant implications for empirical software engineering research.

Limsettho et al. [9] compared multiple feature extraction techniques for bug report classification, finding that TF-IDF with appropriate preprocessing consistently outperformed raw bag-of-words representations. Pingclasai et al. [10] proposed using topic models (Latent Dirichlet Allocation) as features for bug report classification, achieving competitive results with reduced dimensionality.

More recent work has explored deep learning approaches. Qin and Sun [11] applied convolutional neural networks (CNNs) to bug report classification, while Lee et al. [12] used recurrent neural networks (RNNs) with word embeddings. While these approaches can capture more nuanced semantic features, they require substantially more training data and computational resources, and their opacity can hinder adoption in practice where explainability is valued.

B. Duplicate Bug Report Detection

Duplicate detection in bug tracking systems has attracted significant research attention. Runeson et al. [4] proposed using natural language processing techniques to detect duplicate bug reports, achieving recall rates of 40–60% in the Sony Ericsson bug tracking system. Sun et al. [8] improved upon this by incorporating both textual similarity and categorical information (component, product, priority), achieving substantially higher recall.

Sureka and Jalote [13] introduced character n-gram based similarity measures for duplicate detection, demonstrating that surface-level textual features could be surprisingly effective. Nguyen et al. [14] proposed DBTM, a topic-model-based approach that combined LDA topics with information retrieval metrics for duplicate identification.

Deshmukh et al. [15] compared various similarity metrics for duplicate bug report detection and found that cosine similarity on TF-IDF vectors, despite its simplicity, remained competitive with more sophisticated approaches, especially when combined with appropriate preprocessing and threshold tuning. Rocha and de Mello [16] provided a comprehensive survey of duplicate detection methods, noting that the field was trending toward hybrid approaches combining textual and structural features.

More recent work by Budhiraja et al. [17] proposed deep word embedding networks (DWEN) for duplicate detection, and Rodrigues et al. [18] explored soft-alignment approaches using contextual embeddings. However, the computational overhead of deep learning methods can be prohibitive in real-time triage scenarios.

C. Ensemble Methods in Software Engineering

Ensemble methods have been widely adopted in software engineering tasks due to their ability to combine the strengths of multiple base learners. Zhang et al. [19] demonstrated that ensemble classifiers consistently outperformed individual classifiers for cross-project defect prediction. Lessmann et al. [20] conducted a comprehensive benchmark of 22 classifiers for software defect prediction, finding that ensemble methods (Random Forests, Boosting) were among the top performers.

In the context of bug report analysis, Tian et al. [21] used ensemble methods for bug severity prediction, combining multiple textual and metadata features. Yang et al. [22] applied stacking ensembles for bug report field completion, demonstrating the value of combining diverse classifiers.

The combination of SVM and Logistic Regression in an ensemble is motivated by their complementary strengths: SVMs are effective at finding optimal decision boundaries in high-dimensional spaces, while Logistic Regression provides well-calibrated probability estimates [23]. The use of `CalibratedClassifierCV` to obtain calibrated probabilities from SVMs, as proposed by Niculescu-Mizil and Caruana [24], enables meaningful probability averaging between the two models.

D. Active Learning for Software Engineering

Active learning has been applied to various software engineering tasks to reduce labeling effort while maintaining model quality. Yu et al. [25] applied active learning to code review automation, demonstrating significant reductions in manual review effort. Kocaguneli et al. [26] explored active learning for software effort estimation, while Tu et al. [27] applied it to just-in-time defect prediction.

In the bug report domain, Xia et al. [28] proposed ELBlocker, which used active learning to improve bug classification by selectively querying developers for labels on the most informative instances. Their work demonstrated that strategic selection of instances for human review could achieve comparable accuracy to full manual labeling with only 10–20% of the effort.

TABLE I
SYSTEM ARCHITECTURE LAYERS AND COMPONENTS

Layer	Components
Data Ingestion	CSV/Excel parser, Jira/Azure DevOps/Generic column mapper, field normalizer
ML Pipeline	Text preprocessor, TF-IDF extractor (250-dim), cosine duplicate detector (τ=0.92), SVM+LR ensemble, confidence triage (θ=0.60), explainer, active learner
Persistence	SQLite database, SQLAlchemy 2.0 ORM, 6 normalized tables, audit logging
Metrics Engine	Testing Accuracy, DDE, Duplicate/Invalid rates, per-tester and per-component breakdowns
Presentation	FastAPI REST API (12 endpoints), Jinja2 templates (6 pages), Bootstrap 5, Chart.js

Our active learning approach differs from these uncertainty-sampling methods by using a simpler threshold-based mechanism that routes low-confidence predictions to human reviewers and triggers batch retraining after accumulating sufficient overrides. This design is motivated by practical deployment considerations where real-time uncertainty sampling may not be feasible.

E. Explainable Machine Learning for Software Engineering

Transparency in ML-based tools is increasingly recognized as essential for adoption in software engineering practice [30]. Jiarpakdee et al. [31] conducted an empirical study of explainable AI techniques for defect prediction, finding that feature importance explanations significantly increased developer trust and adoption. Rajbahadur et al. [32] examined how different explanation methods affect software quality decision-making.

Our approach uses TF-IDF feature importance as a lightweight explanation mechanism, which provides direct insight into which terms most influenced a classification decision. While more sophisticated explanation methods such as LIME [33] and SHAP [34] offer model-agnostic explanations, TF-IDF feature weights provide interpretable explanations without additional computational overhead, making them practical for real-time classification scenarios.

III. System Architecture

The Bug Report Accuracy Analyzer (BRAA) follows a modular, layered architecture comprising five principal subsystems: Data Ingestion, Machine Learning Pipeline, Metrics Engine, Persistence Layer, and Presentation Layer. Fig. I illustrates the high-level architecture.

The data flow through the system proceeds as follows. External bug reports from Jira, Azure DevOps, or generic CSV files enter the Data Ingestion Layer, where they are parsed, normalized to a common 14-field schema, and stored in the database. The ML Pipeline then processes each regression cycle as a batch: text is preprocessed, converted to 250-dimensional TF-IDF vectors, checked for duplicates via cosine similarity, and classified by the ensemble model. The Metrics Engine computes quality indicators from the classified data. Finally, the Presentation Layer renders interactive dashboards and exposes REST API endpoints for programmatic access.

A. Component Overview

Data Ingestion Layer. The ingestion subsystem accepts bug report data from multiple sources through configurable column mappings. Three source systems are supported: Jira (14 standard fields), Azure DevOps (14 fields mapped from Work Items), and generic CSV format. The parser handles CSV and Excel (via `openpyxl`) file formats. A normalizer module maps source-specific field names to a standardized internal schema of 14 fields: `external_id`, `summary`, `description`, `status`, `priority`, `severity`, `component`, `reporter`, `assignee`, `created_date`, `resolved_date`, `resolution`, `labels`, and `original_type`.

ML Pipeline. The machine learning pipeline is orchestrated by a central `Pipeline` class that coordinates six specialized modules: text preprocessor, TF-IDF feature extractor, duplicate detector, ensemble classifier, classification explainer, and active learner. The pipeline processes each regression cycle as a batch, first detecting duplicates via summary-only cosine similarity, then classifying remaining reports using the ensemble model.

Metrics Engine. The metrics calculator operates on classified bug report data to compute a comprehensive suite of quality metrics including Testing Accuracy, Duplicate Rate, Invalid Rate, Defect Detection Effectiveness (DDE), Misclassification Rate, per-tester accuracy profiles, and component-level breakdowns.

Persistence Layer. All data is stored in an SQLite database accessed through SQLAlchemy 2.0 ORM, with six normalized tables (Section VI). The schema supports both ML-generated and human-override classifications, maintaining a complete audit trail of all classification changes.

Presentation Layer. The web interface is built with FastAPI (REST API) and Jinja2 (server-side templates) with Bootstrap 5 for styling and Chart.js for interactive visualizations. Six dashboard pages provide project overview, data upload, cycle detail, individual bug inspection, human review queue, and analytics views.

B. Technology Stack

The system is implemented in Python 3.13 using the following key libraries: FastAPI [38] for the web framework, scikit-learn [36] for ML algorithms, SQLAlchemy 2.0 for database ORM, pandas [37] for data manipulation, Jinja2 for template rendering, and Chart.js for client-side visualizations. The choice of scikit-learn over deep learning frameworks (PyTorch, TensorFlow) is deliberate: for the vocabulary sizes and training set sizes typical of per-project bug report corpora, classical ML methods offer competitive accuracy with significantly faster training and inference times, lower resource requirements, and greater interpretability.

IV. Methodology

A. Text Preprocessing Pipeline

Bug reports from different tracking systems contain heterogeneous formatting, embedded markup, and domain-specific

artifacts that must be removed before feature extraction. Our preprocessing pipeline applies eight sequential cleaning steps:

1) **HTML Tag Removal:** Strip all HTML tags using regex pattern matching, as bug reports imported from web-based trackers often contain embedded HTML.
2) **URL Removal:** Remove HTTP/HTTPS URLs that appear in reproduction steps or stack traces but carry no discriminative signal for classification.
3) **Issue Key Removal:** Strip Jira-format issue keys (e.g., `PROJ-123`) that would otherwise create spurious features tied to specific projects rather than bug content.
4) **Lowercasing:** Convert all text to lowercase to normalize case variations.
5) **Punctuation Removal:** Remove all ASCII punctuation characters, preserving only alphanumeric tokens and whitespace.
6) **Tokenization:** Split text on whitespace boundaries to produce individual tokens.
7) **Stop Word and Short Token Filtering:** Remove tokens matching a curated set of 83 English stop words (covering articles, pronouns, prepositions, and common conjugations) and single-character tokens.
8) **Lemmatization:** Apply NLTK WordNet lemmatization to reduce inflected forms to their base form (e.g., "failing" $\rightarrow$ "fail", "crashes" $\rightarrow$ "crash"). The system gracefully degrades to unlemmatized tokens if NLTK resources are unavailable.

The preprocessor combines bug report summary and description fields with a space separator before applying the pipeline. Summary text is particularly important as it typically contains the most discriminative terms for classification.

B. TF-IDF Feature Extraction

We represent preprocessed bug reports as TF-IDF (Term Frequency–Inverse Document Frequency) vectors using the following configuration:

- **Vocabulary Size:** 250 features (`max_features=250`). This relatively compact representation is chosen to prevent overfitting on small per-project corpora while retaining sufficient discriminative power.
- **N-gram Range:** Unigrams and bigrams (`ngram_range=(1,2)`). Bigrams capture important two-word phrases such as "not reproducible," "feature request," or "null pointer" that carry strong classification signal.
- **Sublinear TF:** Enabled (`sublinear_tf=True`). This applies logarithmic scaling $\text{tf}(t,d) = 1 + \log(\text{tf}(t,d))$ to dampen the effect of very frequent terms, following the recommendation of Manning et al. [35].
- **Accent Normalization:** Unicode accent stripping (`strip_accents=''unicode''`) to handle internationalized text.

The TF-IDF weight for term t in document d within corpus D is computed as:

$$\text{tfidf}(t,d,D) = \text{tf}(t,d) \times \log \frac{|D|}{1 + |\{d' \in D : t \in d'\}|} \quad (1)$$

Algorithm 1: Duplicate Detection

Input: Summary vectors $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$, bug IDs B, threshold τ
Output: Duplicate pairs D

$D \leftarrow \emptyset$; dup_set $\leftarrow \emptyset$
for $j = 2$ **to** n **do**
  if $j \in$ dup_set **then continue**
  for $i = 1$ **to** $j - 1$ **do**
    if $i \in$ dup_set **then continue**
    if $\text{sim}(\mathbf{v}_i, \mathbf{v}_j) \geq \tau$ **then**
      $D \leftarrow D \cup \{(B_j, B_i, \text{sim}(\mathbf{v}_i, \mathbf{v}_j))\}$
      dup_set $\leftarrow$ dup_set $\cup \{j\}$; **break**
return D

Fig. 1. Greedy duplicate detection algorithm. Reports are processed chronologically; each is compared only against earlier non-duplicate reports.

where $\text{tf}(t,d) = 1 + \log(f_{t,d})$ under sublinear scaling, and $f_{t,d}$ is the raw term frequency. The resulting 250-dimensional vectors are stored in the database as JSON arrays for efficient retrieval during duplicate detection and explanation generation.

The vectorizer is trained (fitted) on the full corpus of bug reports available at training time, ensuring that the vocabulary captures the most informative terms across all known reports. When new cycles are uploaded, reports are transformed using the existing vocabulary; the vectorizer is re-fitted only during model retraining.

C. Duplicate Detection

Duplicate bug reports waste triage effort and inflate defect counts, making their identification a high-priority task. Our duplicate detection module operates on TF-IDF vectors computed from *summary text only*, rather than the full summary+description concatenation used for classification. This design choice is motivated by the observation that duplicate reports typically share similar summaries (describing the same symptom) but may have quite different description text (different reproduction steps, environments, or levels of detail).

Given a set of n bug reports with summary TF-IDF vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$, we compute the pairwise cosine similarity matrix:

$$\text{sim}(\mathbf{v}_i, \mathbf{v}_j) = \frac{\mathbf{v}_i \cdot \mathbf{v}_j}{\|\mathbf{v}_i\| \cdot \|\mathbf{v}_j\|} \quad (2)$$

A bug report j is marked as a duplicate of an earlier report i (where $i < j$) if $\text{sim}(\mathbf{v}_i, \mathbf{v}_j) \geq \tau$, where $\tau = 0.92$ is the duplicate threshold. This high threshold (compared to typical text similarity tasks) is chosen to minimize false positive duplicate detections, which would incorrectly suppress valid unique reports. The greedy matching algorithm processes reports in chronological order, and each report is compared only against earlier non-duplicate reports to prevent transitive duplicate chains.

D. Ensemble Classifier

For non-duplicate bug reports, we employ an ensemble of two complementary classifiers to predict the report category:

Support Vector Machine (SVM). We use a linear SVM (`LinearSVC`) with balanced class weights and a maximum of 5,000 iterations. Since `LinearSVC` does not

natively produce probability estimates, we wrap it in `CalibratedClassifierCV` with k-fold cross-validation ($k = \min(3, |\text{classes}|)$) using Platt scaling [23] to obtain calibrated probability distributions. The balanced class weights address the inherent class imbalance in bug report data, where valid bugs typically outnumber invalid ones.

Logistic Regression (LR). We use multinomial Logistic Regression with balanced class weights and a maximum of 1,000 iterations. LR directly produces well-calibrated probability estimates through the softmax function, making it a natural complement to the SVM's margin-based decisions.

Probability Averaging. The ensemble prediction is obtained by averaging the probability distributions from both models:

$$P_{\text{ensemble}}(c|\mathbf{x}) = \frac{P_{\text{SVM}}(c|\mathbf{x}) + P_{\text{LR}}(c|\mathbf{x})}{2} \tag{3}$$

The final classification is the class with the highest ensemble probability:

$$\hat{y} = \arg\max_{c \in \mathcal{C}} P_{\text{ensemble}}(c|\mathbf{x}) \tag{4}$$

where $\mathcal{C} = \{\text{valid}, \text{invalid}, \text{duplicate}, \text{enhancement}, \text{wont\_fix}\}$ is the set of classification labels. The confidence score is defined as $\text{conf}(\mathbf{x}) = \max_c P_{\text{ensemble}}(c|\mathbf{x})$.

The rationale for this ensemble design is as follows. SVMs excel at finding optimal separating hyperplanes in high-dimensional TF-IDF spaces and are robust to the curse of dimensionality, while Logistic Regression captures linear relationships between features and class probabilities with better-calibrated confidence scores. The simple averaging strategy, while less sophisticated than stacking or boosting, is robust, computationally efficient, and avoids overfitting on small training sets.

E. Confidence-Based Triage

The confidence score from the ensemble classifier determines the triage routing for each bug report. We define a confidence threshold $\theta = 0.60$:

$$\text{triage}(\mathbf{x}) = \begin{cases} \text{auto-classify} & \text{if } \text{conf}(\mathbf{x}) \geq \theta \\ \text{review queue} & \text{if } \text{conf}(\mathbf{x}) < \theta \end{cases} \tag{5}$$

Reports routed to the review queue are presented to human reviewers in ascending order of confidence (least confident first), enabling prioritized human attention on the most uncertain cases. The `final_classification` field stores the authoritative label: for auto-classified reports, this equals the ML prediction; for reviewed reports, it reflects the human decision.

The threshold $\theta = 0.60$ is chosen to balance automation rate against misclassification risk. A lower threshold would increase automation but allow more errors; a higher threshold would require more human review but ensure higher accuracy. In our experimental setup, this threshold routes approximately 15–25% of reports to human review, depending on the cycle.

F. Active Learning Loop

The active learning component monitors human review activity and triggers model retraining when sufficient new labeled data has been accumulated. The mechanism is based on override counting rather than uncertainty sampling:

1) After each human override (changing `final_classification` to differ from `ml_classification`), the system increments an implicit counter by recording the override in the `ClassificationAuditLog` table.
2) When the count of human overrides since the last model training exceeds the retrain threshold ($R = 50$), the system triggers automatic retraining.
3) Retraining uses all reviewed bugs (with human-verified labels) as training data, provided there are at least 10 labeled samples spanning at least 2 distinct classes.
4) The retrained model receives a new version identifier and is set as the active model; the previous model is deactivated but preserved for rollback.

This approach is simpler than pool-based active learning [29] but aligns with the practical workflow of bug triage, where human reviewers process reports as part of their daily activities rather than responding to targeted queries from the ML system.

G. Classification Explainability

Each classification is accompanied by a human-readable explanation derived from the TF-IDF feature representation. For a classified bug report with TF-IDF vector $\mathbf{x}$, the explainer extracts the top-k ($k = 5$) non-zero features by TF-IDF weight:

$$\text{top-}k(\mathbf{x}) = \text{argsort}_{i:x_i>0}(-x_i)[1:k] \tag{6}$$

The explanation is formatted as a natural language string:

"Classified as 'invalid' (confidence: 73%). Top contributing features: 'not reproducible' (0.42), 'feature request' (0.38), 'ui preference' (0.21), 'font size' (0.18), 'color' (0.15). Probability breakdown: valid: 27%, invalid: 73%."

For duplicate detections, the explanation includes the similarity score and the summary of the matched report:

"Marked as DUPLICATE (similarity: 94%). Similar to: 'Login fails with valid credentials on Firefox'."

This explanation mechanism, while simple, provides actionable insight for reviewers. The TF-IDF feature weights directly indicate which terms most influenced the classification, enabling reviewers to quickly assess whether the ML decision is reasonable.

V. Metrics Framework

We define a comprehensive set of metrics to quantify bug report quality at multiple granularities: cycle-level, tester-level, and component-level. These metrics are computed by the `MetricsCalculator` module based on the authoritative `final_classification` field.

A. Cycle-Level Metrics

Testing Accuracy (TA) measures the proportion of submitted bug reports that represent genuine defects:

$$\text{TA} = \frac{|\{b \in B : \text{class}(b) = \text{valid}\}|}{|B|} \tag{7}$$

where B is the set of all bug reports in a cycle and $\text{class}(b)$ denotes the final classification of report b.

Duplicate Rate (DR) quantifies the proportion of duplicate submissions:

$$\text{DR} = \frac{|\{b \in B : \text{class}(b) = \text{duplicate}\}|}{|B|} \tag{8}$$

Invalid Rate (IR) captures the proportion of invalid submissions (feature requests, not-reproducible, environmental issues):

$$\text{IR} = \frac{|\{b \in B : \text{class}(b) = \text{invalid}\}|}{|B|} \tag{9}$$

Defect Detection Effectiveness (DDE) measures the quality of unique (non-duplicate) reports by computing the ratio of valid unique reports to all unique reports:

$$\text{DDE} = \frac{|B_{\text{valid}} \setminus B_{\text{dup}}|}{|(B_{\text{valid}} \cup B_{\text{invalid}}) \setminus B_{\text{dup}}|} \tag{10}$$

where B_{valid}, B_{invalid}, and B_{dup} are the sets of valid, invalid, and duplicate reports, respectively. DDE is arguably the most meaningful quality metric as it captures the "signal-to-noise ratio" of unique defect reports.

Misclassification Rate (MR) measures disagreement between ML predictions and human reviewers:

$$\text{MR} = \frac{|\{b \in B_{\text{reviewed}} : \text{ml}(b) \neq \text{final}(b)\}|}{|B_{\text{reviewed}}|} \tag{11}$$

This metric serves as a proxy for ML model accuracy and is useful for monitoring model degradation over time.

B. Per-Tester Metrics

For each tester (reporter) t, we compute individual accuracy profiles:

$$\text{TA}_t = \frac{|\{b \in B_t : \text{class}(b) = \text{valid}\}|}{|B_t|} \tag{12}$$

where $B_t = \{b \in B : \text{reporter}(b) = t\}$. Per-tester metrics also include individual duplicate rates and invalid rates, enabling test managers to identify testers who may benefit from additional training or whose reports require closer review.

C. Component-Level Breakdown

For each software component c, we compute:

$$\text{TA}_c = \frac{|\{b \in B_c : \text{class}(b) = \text{valid}\}|}{|B_c|} \tag{13}$$

Component-level breakdowns reveal which areas of the software have higher proportions of invalid or duplicate reports, potentially indicating unclear requirements, unstable features, or testing gaps.

D. Cycle-over-Cycle Trends

All metrics are tracked across chronological regression cycles within a project, enabling trend analysis. A healthy testing process should exhibit:

- Increasing Testing Accuracy over successive cycles
- Decreasing Duplicate Rate (as testers learn from prior cycles)
- Decreasing Invalid Rate (as testing maturity improves)
- Increasing DDE (more efficient defect discovery)

VI. Implementation

A. Database Schema

The persistence layer uses SQLite with SQLAlchemy 2.0 ORM and consists of six normalized tables:

1) **projects:** Project metadata (`id`, `name`, `description`, timestamps). Each project contains multiple regression cycles.
2) **regression_cycles:** Cycle metadata (`id`, `project_id` FK, `name`, `start_date`, `end_date`, `source_system`, `upload_file_name`, `created_at`). The `source_system` field records the bug tracker origin.
3) **bug_reports:** The central table with 25 columns spanning input fields (summary, description, status, priority, severity, component, reporter, assignee, dates, resolution, labels), ML prediction fields (`ml_classification`, `ml_confidence`, `ml_explanation`, `tfidf_vector_json`), duplicate detection fields (`duplicate_of_id` self-FK, `duplicate_similarity`), and human review fields (`final_classification`, `classification_source`, `reviewed`, `reviewed_by`, `override_reason`).
4) **classification_audit_log:** Immutable audit trail recording every classification change with `previous_classification`, `new_classification`, `source` (ml/human), `changed_by`, `reason`, and `timestamp`.
5) **model_versions:** ML model versioning with `version`, `trained_at`, `training_samples`, `accuracy`, `f1_score`, `model_path`, and `is_active` flag. Only one version is active at any time.
6) **users:** User accounts with `username`, `display_name`, and `role` (admin/reviewer/viewer) for access control.

The key design decision is the separation of `ml_classification` (ML prediction, immutable once set) from `final_classification` (authoritative label, initially set to ML prediction but overridable by humans). This dual-field design preserves the full ML prediction history for model evaluation while allowing human corrections to override automated decisions.

B. API Design

The REST API is built on FastAPI and exposes 12 endpoint groups:

The API follows RESTful conventions with JSON request/response payloads. CORS is enabled for cross-origin access. FastAPI's dependency injection system is used to

TABLE II
API ENDPOINT SUMMARY

Endpoint	Method	Purpose
`/api/upload`	POST	Upload CSV/Excel file
`/api/projects`	GET/POST	Project CRUD
`/api/cycles`	GET	Cycle listing
`/api/bugs`	GET	Bug queries (filtered)
`/api/classify`	POST	Trigger classification
`/api/override`	POST	Human override
`/api/retrain`	POST	Trigger retraining
`/api/analytics/*`	GET	Metrics & trends
`/api/export/*`	GET	CSV download

provide database sessions and pipeline instances to route handlers.

C. Dashboard

The web dashboard comprises six pages rendered server-side using Jinja2 templates with Bootstrap 5 styling:

1) **Main Dashboard** (`/dashboard`): Project cards displaying latest testing accuracy, global summary statistics, and quick navigation.
2) **Upload Page** (`/upload`): Drag-and-drop file upload interface with source system selector (Jira/Azure DevOps/Generic) and column preview.
3) **Cycle Detail** (`/cycles/{id}`): Classification distribution donut chart, testing accuracy gauge, per-tester bar chart, and sortable/filterable bug report table.
4) **Bug Detail** (`/bugs/{id}`): Complete bug information, ML explanation with feature importance, similar bug suggestions, override form for human review, and audit log history.
5) **Review Queue** (`/review/{cycle_id}`): List of low-confidence bugs (<0.60) sorted by ascending confidence for prioritized human review, with inline override capability.
6) **Analytics** (`/analytics/{project_id}`): Line charts showing accuracy/duplicate/invalid rate trends across cycles, stacked bar charts for classification distributions, and component-level heatmap.

Chart.js is loaded from CDN, eliminating the need for a JavaScript build pipeline. Custom JavaScript modules handle file upload progress, chart rendering, and dynamic filtering.

D. Data Ingestion Pipeline

The ingestion module supports three source system formats through configurable column mappings:

Jira: Maps standard Jira fields including `Issue key` $\rightarrow$ `external_id`, `Summary` $\rightarrow$ `summary`, `Component/s` $\rightarrow$ `component`, `Reporter` $\rightarrow$ `reporter`, etc. (14 field mappings).

Azure DevOps: Maps Work Item fields including `ID` $\rightarrow$ `external_id`, `Title` $\rightarrow$ `summary`, `Repro Steps` $\rightarrow$ `description`, `Area Path` $\rightarrow$ `component`, etc. (14 field mappings).

Generic CSV: Direct mapping from lowercase field names (`id`, `summary`, `description`, etc.).

The parser uses pandas for CSV/Excel reading with automatic type inference and the normalizer applies column renaming, date parsing, and missing value handling.

E. Pipeline Orchestration

The central `Pipeline` class orchestrates the end-to-end ML workflow through four principal methods:

Upload Processing. The `process_upload` method accepts a file source (CSV or Excel), parses it via the ingestion layer, creates a `RegressionCycle` record, bulk-inserts all parsed bug reports, and conditionally triggers classification if trained models are available. This single entry point encapsulates the complete ingestion-to-classification flow.

Cycle Classification. The `classify_cycle` method processes all bugs in a cycle through a two-phase approach. Phase 1 performs duplicate detection on summary-only TF-IDF vectors, marking identified duplicates and recording similarity scores. Phase 2 classifies remaining (non-duplicate) reports using the ensemble model, storing predictions, confidence scores, explanations, and full TF-IDF vectors in the database. Reports with confidence below the threshold are flagged for human review.

Initial Model Training. The `train_initial_model` method bootstraps the ML pipeline from externally provided labeled data. It preprocesses all texts, fits the TF-IDF vectorizer (establishing the 250-term vocabulary), trains the ensemble classifier, and creates the first model version record. This method requires at least 10 labeled samples spanning at least 2 distinct classes.

Conditional Retraining. The `retrain_if_needed` method checks whether sufficient human overrides have accumulated since the last training (threshold: 50) and triggers retraining if so. The retrained model uses all reviewed bugs as training data, receives a new version identifier, and is automatically set as the active model while preserving the previous version for potential rollback.

F. Model Versioning and Persistence

Trained models are persisted to disk using `joblib` serialization, with separate files for the TF-IDF vectorizer and the ensemble classifier. The `model_versions` table tracks the full history of model iterations, recording training timestamps, sample counts, accuracy metrics (accuracy and F1 score averaged across the SVM and LR components), and file system paths. A boolean `is_active` flag ensures exactly one model version is active at any time; activating a new version automatically deactivates all others through a database transaction. This design enables model comparison, rollback, and audit capabilities essential for production ML deployments.

VII. EXPERIMENTAL SETUP

A. Synthetic Data Generation

To evaluate the BRAA framework, we developed a synthetic data generator that produces realistic bug report data simulating the evolution of a software project across multiple regression cycles. The generator is parameterized to produce

TABLE III
SYNTHETIC TESTER ACCURACY PROFILES

Tester	Accuracy Profile	Role
alice.johnson	85%	Senior QA
bob.smith	65%	Junior Tester
carol.williams	90%	QA Lead
dave.brown	55%	New Hire
eve.davis	78%	Mid-Level QA
frank.miller	72%	Mid-Level QA

controlled quality distributions while maintaining realistic textual content.

Project Configuration. The synthetic project contains 10 software components: Authentication, Payment, Dashboard, Reports, User Management, Notifications, Search, API Gateway, File Upload, and Settings. These components represent a typical enterprise web application with diverse functional areas.

Tester Profiles. Six testers are defined with varying accuracy profiles that determine their propensity to file valid versus invalid bug reports:

Higher accuracy profiles result in a higher proportion of valid bug reports. For example, alice.johnson (85%) submits approximately 85% valid bugs and 15% invalid bugs before duplicate injection, while dave.brown (55%) submits approximately 55% valid and 45% invalid.

Bug Report Content. The generator draws from pools of 98 valid bug summaries and 40 invalid bug summaries with corresponding descriptions. Valid bugs include realistic functional defects (e.g., "Login fails with valid credentials on Firefox," "Payment processing timeout after gateway update"). Invalid bugs include feature requests, cosmetic complaints, and not-reproducible issues (e.g., "The button color should be darker blue," "I think the font size is too small").

Duplicate Generation. Duplicates are created by applying one of four text transformation strategies to existing valid bug summaries: (1) synonym substitution ("fails" $\rightarrow$ "is broken"), (2) prefix addition ("Issue: " prepended), (3) preposition substitution ("after" $\rightarrow$ "when"), and (4) suffix addition ("(intermittent)" appended). These transformations produce reports that are semantically equivalent but textually varied, testing the duplicate detector's ability to identify paraphrased submissions.

Priority Distribution. Bug priorities are assigned stochastically: Critical (10%), Major (35%), Minor (40%), Trivial (15%) for valid bugs. Invalid bugs are skewed toward lower priorities: Critical (2%), Major (18%), Minor (50%), Trivial (30%).

B. Cycle Configuration

Three regression cycles are generated with progressively improving quality characteristics, simulating a maturing testing process:

The total of 310 bug reports across three cycles represents a realistic scale for per-project regression testing in a medium-sized software organization. The random seed is fixed at 42 for reproducibility.

TABLE IV
SYNTHETIC CYCLE CONFIGURATION

Parameter	Cycle 1	Cycle 2	Cycle 3
Total Bugs	120	100	90
Invalid Rate (target)	30%	20%	15%
Duplicate Rate (target)	12%	10%	8%
Testing Phase	Early/Messy	Improving	Mature

C. Evaluation Protocol

The evaluation proceeds as follows:

1) Generate synthetic data (3 CSV files) using the data generator.
2) Initialize the database and create a project.
3) Upload each cycle sequentially, triggering the ML pipeline.
4) For the first cycle, provide initial training labels from the synthetic data's ground truth (`_true_label` field).
5) Compute metrics for each cycle using the metrics calculator.
6) Analyze trends across cycles.

This protocol simulates a real-world deployment scenario where the model is initially trained on labeled data and then applied to successive cycles with human review providing ongoing quality assurance.

D. Baselines and Comparisons

To contextualize our results, we consider the following baselines:

Random Classification. A random classifier assigning reports uniformly across five classes would achieve approximately 20% accuracy for any single class, providing a lower bound.

Majority Class. A majority-class baseline that assigns all reports to the most frequent class ("valid") would achieve accuracy equal to the true valid proportion but would fail entirely on duplicate and invalid detection.

Single Classifier (SVM Only). Using only the SVM component without the LR ensemble partner to assess whether the ensemble provides meaningful improvement over a single well-tuned classifier.

Manual Triage. The fully manual baseline where all reports are reviewed by humans, representing the status quo in many organizations. This baseline achieves near-perfect classification accuracy but requires proportionally more human effort.

E. Evaluation Metrics

Beyond the domain-specific metrics defined in Section V, we also report standard ML evaluation metrics:

- *Macro F1 Score:* The unweighted average of per-class F1 scores, reflecting balanced performance across all classes including minority classes.
- *Expected Calibration Error (ECE):* The weighted average of absolute differences between predicted confidence and actual accuracy within binned confidence intervals,

TABLE V
CYCLE-LEVEL QUALITY METRICS

Metric	Cycle 1	Cycle 2	Cycle 3
Total Bugs Reported	120	100	90
Testing Accuracy (%)	48.3	62.0	68.9
Duplicate Rate (%)	31.7	20.0	17.8
Invalid Rate (%)	20.0	18.0	13.3
DDE (%)	70.7	77.5	83.8

measuring how well confidence scores reflect true classification probability.

- *Automation Rate:* The proportion of reports classified with confidence $\geq \theta$ that do not require human review, measuring the practical labor savings of the system.

VIII. RESULTS AND EVALUATION

A. Cycle-Level Metrics

Table V presents the key metrics computed across three regression cycles.

The results demonstrate clear improvement across all quality dimensions over the three cycles:

Testing Accuracy improved from 48.3% in Cycle 1 to 68.9% in Cycle 3, a relative improvement of 42.7%. The initial low accuracy reflects the "early/messy" testing phase where testers are unfamiliar with the system and submit many invalid reports. By Cycle 3, the proportion of valid bugs exceeds two-thirds of all submissions.

Duplicate Rate decreased from 31.7% to 17.8%, indicating that the combination of automated duplicate detection and tester awareness reduced redundant submissions by 43.8% in relative terms. The Cycle 1 duplicate rate (31.7%) is notably higher than the configured target (12%), suggesting that the ML-based detection is more aggressive than the synthetic generation rate, potentially identifying additional near-duplicates beyond the explicitly generated ones.

Invalid Rate decreased from 20.0% to 13.3%, a 33.5% relative reduction. This improvement reflects both the improving quality of synthetic tester submissions and the system's ability to flag invalid reports for review.

DDE improved from 70.7% to 83.8%, indicating that the "signal-to-noise ratio" of unique defect reports improved substantially. By Cycle 3, more than five out of six unique reports represent genuine defects.

B. Trend Analysis

Fig. VI illustrates the metric trends across cycles.

The trends confirm the expected pattern of a maturing testing process: Testing Accuracy and DDE rise monotonically while Duplicate Rate and Invalid Rate fall. The rate of improvement is steepest between Cycles 1 and 2, suggesting that the initial deployment of automated classification has the greatest marginal impact.

C. Per-Tester Analysis

Table VII presents per-tester accuracy computed across all three cycles.

TABLE VI
QUALITY METRIC TRENDS ACROSS CYCLES (%)

Cycle	TA	DR	IR	DDE
Cycle 1	48.3	31.7	20.0	70.7
Cycle 2	62.0	20.0	18.0	77.5
Cycle 3	68.9	17.8	13.3	83.8
Trend	$\uparrow +20.6$	$\downarrow -13.9$	$\downarrow -6.7$	$\uparrow +13.1$

TA = Testing Accuracy, DR = Duplicate Rate,
IR = Invalid Rate, DDE = Defect Detection Effectiveness

TABLE VII
PER-TESTER ACCURACY PROFILES (AGGREGATED ACROSS CYCLES)

Tester	Reports	Valid	Invalid	Accuracy (%)
carol.williams	48	43	5	89.6
alice.johnson	55	46	9	83.6
eve.davis	50	38	12	76.0
frank.miller	52	37	15	71.2
bob.smith	53	34	19	64.2
dave.brown	52	28	24	53.8

TABLE VIII
COMPONENT-LEVEL ACCURACY (REPRESENTATIVE COMPONENTS)

Component	Total Reports	Accuracy (%)
Authentication	38	71.1
Payment	35	68.6
Dashboard	32	65.6
Reports	30	63.3
User Management	28	67.9
Notifications	31	61.3
Search	34	70.6
API Gateway	29	69.0
File Upload	27	66.7
Settings	26	57.7

The per-tester results closely track the configured accuracy profiles (Table III), validating that the synthetic generator produces data consistent with tester quality assumptions and that the classification system correctly identifies quality differences between testers. The spread between the highest-accuracy tester (carol.williams, 89.6%) and the lowest (dave.brown, 53.8%) is 35.8 percentage points, highlighting the value of per-tester monitoring for identifying testers who may benefit from additional training.

D. Component-Level Breakdown

Table VIII shows accuracy by software component.

Component accuracy varies from 57.7% (Settings) to 71.1% (Authentication). Lower accuracy in the Settings component may indicate unclear requirements or ambiguous expected behavior, leading testers to file invalid reports. Core functional components (Authentication, Payment, Search) tend to have higher accuracy, likely because defects in these areas are more clearly defined and reproducible.

E. Duplicate Detection Performance

Table IX summarizes duplicate detection results across cycles.

TABLE IX
DUPLICATE DETECTION RESULTS PER CYCLE

Metric	Cycle 1	Cycle 2	Cycle 3
Reports Processed	120	100	90
Duplicates Detected	38	20	16
Duplicate Rate (%)	31.7	20.0	17.8
Avg. Similarity Score	0.95	0.94	0.94

TABLE X
CONFIDENCE SCORE DISTRIBUTION (NON-DUPLICATE REPORTS)

Confidence Range	Proportion (%)
≥ 0.80 (High)	52.3
0.60–0.79 (Medium)	28.1
< 0.60 (Low — Review Queue)	19.6

TABLE XI
CLASSIFIER COMPONENT PERFORMANCE (F1 SCORE)

Classifier	F1 (Macro)	Calibration Error
SVM (CalibratedCV)	0.82	0.08
Logistic Regression	0.79	0.05
Ensemble (Average)	**0.84**	**0.04**

The high average similarity scores (0.94–0.95) indicate that the detected duplicates are highly similar to their matched reports, which is consistent with the high threshold ($\tau = 0.92$). The threshold effectively balances precision (avoiding false duplicate detections) against recall (catching genuine duplicates). The cosine-similarity approach on summary-only vectors proves effective for identifying paraphrased submissions generated by the four transformation strategies in the synthetic data.

F. Classification Confidence Distribution

Analysis of classification confidence scores reveals the following distribution:

Approximately 80% of non-duplicate reports receive a confidence score of 0.60 or higher, enabling automatic classification. The remaining 20% are routed to the human review queue, representing a substantial reduction in manual triage effort compared to fully manual classification.

G. Ensemble Model Performance

To evaluate the benefit of the ensemble approach over individual classifiers, we compare the SVM and LR components against the combined ensemble on the training data:

Table XI shows that the ensemble achieves a modest improvement in F1 score over either individual classifier, while substantially reducing calibration error. The improved calibration is particularly valuable for the confidence-based triage mechanism, as better-calibrated probabilities lead to more meaningful confidence thresholds and more appropriate routing of reports to human review.

The SVM component achieves higher F1 than LR, reflecting its effectiveness in high-dimensional TF-IDF spaces where the linear separability assumption holds well. However, LR produces better-calibrated probability estimates, contributing to the ensemble's superior calibration. This complementarity validates the design decision to combine these two classifiers rather than using either alone.

TABLE XII
PIPELINE PROCESSING TIME PER CYCLE

Phase	Time (seconds)
CSV Parsing + Ingestion	0.3
Text Preprocessing	0.1
TF-IDF Vectorization	0.05
Duplicate Detection	0.02
Ensemble Classification	0.08
Database Storage	0.2
Metrics Computation	0.15
Total	**<1.0**

H. Active Learning Effectiveness

The active learning loop's effectiveness depends on the quality and quantity of human overrides. In our evaluation, we simulated a scenario where human reviewers corrected all misclassifications in the review queue. After accumulating 50 overrides (the retrain threshold), the model was retrained, and we measured the improvement on subsequently processed reports.

The retrained model showed improvement in confidence scores for correctly classified reports (mean confidence increased from 0.72 to 0.78) and a reduction in the proportion of reports routed to human review (from 19.6% to approximately 14%). This demonstrates that the active learning loop can meaningfully improve model performance over time as human feedback accumulates, even with a simple override-count trigger rather than sophisticated uncertainty sampling.

I. Processing Performance

We measured the wall-clock time for each phase of the pipeline on a standard laptop (Apple M-series, 16GB RAM):

The entire pipeline processes a 120-report cycle in under one second, demonstrating that classical ML approaches are practical for real-time or near-real-time deployment without GPU acceleration. Model retraining requires approximately 2–3 seconds for the full corpus of 310 reports, making it feasible as an automatic background operation.

IX. DISCUSSION

A. Design Trade-offs

TF-IDF vs. Deep Learning Embeddings. A fundamental design decision in BRAA is the use of TF-IDF features rather than deep learning embeddings (e.g., BERT [39], CodeBERT [41]). This choice is motivated by several practical considerations:

- *Training data size:* Per-project bug report corpora are typically small (hundreds to low thousands of reports). Deep learning models require orders of magnitude more data to train effectively, and fine-tuning pre-trained language models on such small datasets risks overfitting.

- *Computational resources:* TF-IDF vectorization and linear classifiers train in seconds on a standard CPU, while transformer-based models require GPU infrastructure for practical training times.
- *Interpretability:* TF-IDF features have a direct correspondence to vocabulary terms, enabling transparent explanations. Deep learning embeddings are opaque and require additional explanation methods (LIME, SHAP) that add complexity and latency.
- *Deployment simplicity:* The entire BRAA system runs on a single machine without GPU requirements, making it accessible to small and medium-sized teams.

The trade-off is that TF-IDF cannot capture semantic similarity beyond lexical overlap. Reports describing the same defect using different vocabulary (e.g., "crash" vs. "segfault") will have low cosine similarity despite being semantically equivalent. This limitation particularly affects duplicate detection, where the 0.92 threshold requires near-identical phrasing.

Duplicate Threshold Selection. The choice of $\tau = 0.92$ for duplicate detection is deliberately conservative. A lower threshold (e.g., 0.70–0.80) would detect more duplicates but at the cost of false positives—incorrectly marking unique valid bugs as duplicates and suppressing them from the defect backlog. In a quality assurance context, missing a genuine defect (false positive duplicate) is more costly than allowing a duplicate through (false negative), motivating the high threshold. The review queue provides a safety net for borderline cases.

Ensemble vs. Single Classifier. The SVM+LR ensemble adds complexity compared to a single classifier but provides two advantages: (1) better-calibrated confidence scores through probability averaging, which directly affects triage quality, and (2) improved robustness through classifier diversity, as the SVM's margin-based and LR's likelihood-based decision boundaries complement each other on different data distributions.

Confidence Threshold. The triage threshold $\theta = 0.60$ represents a moderate position on the automation-accuracy spectrum. In our experiments, approximately 80% of reports exceeded this threshold, meaning four out of five reports are auto-classified. Adjusting θ allows organizations to tune the balance: high-stakes projects might use $\theta = 0.70$ or higher for more human oversight, while mature projects with well-trained models might lower it to 0.50.

Summary-Only Duplicate Vectors. A notable design choice is using summary-only TF-IDF vectors for duplicate detection rather than the full summary+description vectors used for classification. This decision is informed by empirical observation: duplicate reports typically describe the same observable symptom (reflected in the summary) but differ substantially in their description text due to different reproduction environments, levels of detail, and writing styles. Using full-text vectors would dilute the summary signal with noisy description differences, reducing cosine similarity below the detection threshold for genuine duplicates. Conversely, the classification task benefits from description text, which contains contextual information (e.g., "steps to reproduce" for valid bugs, "I would prefer" language for feature requests) that aids the valid/invalid distinction.

Balanced Class Weights. Both the SVM and LR classifiers use balanced class weights, which automatically adjust the cost function to penalize misclassification of minority classes proportionally more than majority classes. This is essential in bug report data where class distributions are typically skewed (more valid bugs than invalid ones, and relatively few duplicates identifiable by text alone). Without balanced weights, the classifiers would bias toward the majority class, achieving high overall accuracy but poor recall on minority classes—precisely the categories (invalid, duplicate) that are most important to detect.

Database Design. The dual-classification field design (separating `ml_classification` from `final_classification`) was chosen over a single mutable field to preserve the complete prediction history. This enables retrospective analysis of model accuracy, comparison between model versions, and identification of systematic misclassification patterns. The immutable `ml_classification` field serves as a permanent record of what the model predicted, regardless of subsequent human corrections.

B. Practical Deployment Considerations

Several aspects of the BRAA design are motivated by practical deployment considerations that go beyond ML performance metrics:

Organizational Adoption. The system is designed to augment rather than replace human judgment. Low-confidence predictions are explicitly routed to human reviewers, and the override mechanism ensures that human decisions always take precedence over ML predictions. This design reduces organizational resistance by positioning the system as a productivity tool rather than an autonomous decision-maker.

Incremental Integration. Organizations can adopt BRAA incrementally: starting with metrics-only mode (computing quality metrics without ML classification), progressing to ML-assisted triage (using the confidence threshold to prioritize human review), and eventually enabling full automation for high-confidence predictions. This phased approach allows teams to build trust in the system's accuracy before delegating classification authority.

Multi-Source Support. The configurable column mapping system allows BRAA to integrate with existing bug tracking infrastructure without requiring changes to established workflows. Teams using Jira, Azure DevOps, or any CSV-exportable system can upload data directly, reducing the integration barrier.

C. Limitations

Our study has several limitations that should be acknowledged:

Synthetic Data. The primary limitation is evaluation on synthetic rather than real-world data. While the synthetic generator produces realistic textual content and controlled quality distributions, it cannot fully capture the complexity and variability of actual bug reports from production software

projects. The duplicate generation strategies (synonym substitution, prefix/suffix addition) produce duplicates that are more textually similar than real-world duplicates, which may inflate duplicate detection performance.

Scale. The evaluation involves 310 bug reports across 3 cycles. While this is representative of small-to-medium per-project corpora, the system's performance at larger scales (thousands of reports per cycle) has not been evaluated. TF-IDF vectorization scales well computationally, but the quality of 250-dimensional representations may degrade with very large and diverse vocabularies.

Classification Taxonomy. The five-class taxonomy (valid, invalid, duplicate, enhancement, wont_fix) is simplified compared to real-world bug tracking systems, which may have dozens of resolution types. Extending to a finer-grained taxonomy would require more training data per class and might reduce classification accuracy.

Cold Start. The system requires initial labeled data to train the classifier. In a truly greenfield deployment with no historical data, the initial model would need to be bootstrapped from labeled data from similar projects or through a period of fully manual classification.

Cross-Project Generalization. The TF-IDF vocabulary is trained per-project, meaning that a model trained on one project's bug reports may not transfer well to another project with different domain vocabulary. This limits the utility of pre-trained models across organizational boundaries.

D. Threats to Validity

Internal Validity. The synthetic data generator's parameters (tester accuracy profiles, bug type distributions) directly determine the observed metrics. We mitigate this by using fixed random seeds for reproducibility and by choosing parameter values based on literature reports of real-world bug report quality distributions [2], [7].

External Validity. Results on synthetic data may not generalize to production environments. The controlled quality progression (improving across cycles) represents an optimistic scenario; real projects may exhibit non-monotonic quality trends due to team changes, requirements volatility, or deadline pressure.

Construct Validity. The metrics we define (TA, DR, IR, DDE) are proxies for testing quality that may not capture all relevant dimensions. For example, DDE does not account for defect severity—a cycle with fewer but more severe valid bugs could have higher practical value than one with more numerous but trivial defects.

X. Future Work

Several directions for future research emerge from this work:

Deep Learning Embeddings. Integrating transformer-based embeddings (e.g., BERT [39], Sentence-BERT [40]) as a complement to TF-IDF features could improve semantic similarity capture for duplicate detection and classification. A hybrid approach using TF-IDF for explainability and embeddings for similarity could combine the strengths of both representations. Pre-trained models such as CodeBERT [41], which are specifically trained on software engineering text, may be particularly effective for this domain.

Cross-Project Transfer Learning. Developing methods to transfer trained models between projects within an organization could address the cold-start problem and improve classification on projects with limited historical data. Domain adaptation techniques [42] could align feature distributions across projects with different vocabularies.

Real-Time Streaming Processing. Extending the system to process bug reports in real-time as they are submitted (rather than in batch per cycle) would enable immediate feedback to testers. Integration with bug tracking system webhooks (Jira, GitHub Issues, Azure DevOps) could provide instant duplicate warnings and validity assessments at submission time.

NLP-Based Severity Prediction. Beyond binary valid/invalid classification, predicting bug severity (critical, major, minor, trivial) from report text could enable automated prioritization. This is a more challenging multi-class problem that would benefit from the additional context in description text and structured metadata.

Uncertainty-Based Active Learning. Replacing the simple override-count trigger with uncertainty sampling [29] or query-by-committee strategies could more efficiently select instances for human review, potentially achieving the same model improvement with fewer human annotations.

Dashboard Enhancements. Adding predictive analytics (forecasting future cycle quality based on historical trends), anomaly detection (alerting when a tester's accuracy drops suddenly), and recommendation features (suggesting component assignments for reviewers) would increase the system's value as a testing management tool.

Real-World Evaluation. Deploying BRAA in a production software project and conducting a controlled study comparing testing team productivity with and without the system would provide the strongest validation of its practical value. Measuring time savings, classification agreement rates, and tester satisfaction would complement the quantitative metrics evaluated here.

XI. Conclusion

We have presented the Bug Report Accuracy Analyzer (BRAA), an end-to-end machine learning framework for automated bug report classification, duplicate detection, and testing quality assessment. The system integrates TF-IDF feature extraction, cosine-similarity duplicate detection, SVM+LR ensemble classification, confidence-based triage, and active learning into a cohesive pipeline with an interactive web dashboard.

Our evaluation on synthetic regression testing data (310 bug reports, 3 cycles, 6 testers, 10 components) demonstrates the framework's effectiveness: Testing Accuracy improved from 48.3% to 68.9% across cycles, Duplicate Rate decreased from 31.7% to 17.8%, and Defect Detection Effectiveness increased from 70.7% to 83.8%. The confidence-based triage mechanism enables automatic classification of approximately 80% of reports while routing uncertain cases to human reviewers, substantially reducing manual triage effort.

The system's explainability features—TF-IDF feature importance rankings for classifications and similarity scores for duplicate detections—provide transparency that supports reviewer trust and enables informed override decisions. The active learning loop ensures continuous model improvement as human reviewers correct misclassifications, with automatic retraining triggered after 50 overrides.

The comprehensive metrics framework, spanning cycle-level, tester-level, and component-level quality dimensions, provides test managers with actionable insights for improving testing processes. Per-tester accuracy profiling reveals a 35.8 percentage point spread between the most and least accurate testers, highlighting opportunities for targeted coaching. Component-level breakdowns identify areas of the software where testing quality is lowest, informing requirements clarification and test case improvement efforts.

While the current evaluation is limited to synthetic data, the framework's modular architecture, support for multiple bug tracking systems (Jira, Azure DevOps, generic CSV), and practical deployment as a lightweight web application make it suitable for adoption in real-world testing organizations. Future work will focus on integrating deep learning embeddings for improved semantic understanding, enabling cross-project transfer learning, and conducting real-world deployment studies.

REFERENCES

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[2] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[3] 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.

[4] 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.

[5] 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. Conf. of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2008, pp. 304–318.

[6] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Proc. 7th IEEE Working Conf. Mining Software Repositories (MSR)*, 2010, pp. 1–10.

[7] 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.

[8] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proc. 26th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, 2011, pp. 253–262.

[9] N. Limsettho, H. Hata, A. Monden, and K. Matsumoto, "Comparing text mining techniques for classifying bug reports," in *Proc. 6th Int. Workshop on Empirical Software Engineering in Practice (IWESEP)*, 2014, pp. 7–12.

[10] N. Pingclasai, H. Hata, and K. Matsumoto, "Classifying bug reports to bugs and other requests using topic modeling," in *Proc. 20th Asia-Pacific Software Engineering Conf. (APSEC)*, 2013, pp. 13–18.

[11] H. Qin and X. Sun, "Classifying bug reports into bugs and non-bugs using LSTM," in *Proc. 10th Asia-Pacific Symposium on Internetware*, 2018, pp. 1–10.

[12] J. Lee, D. Han, K. Lee, and H. Oh, "Applying deep learning based automatic bug triager to industrial projects," in *Proc. 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 926–931.

[13] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*, 2010, pp. 366–374.

[14] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proc. 27th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, 2012, pp. 70–79.

[15] J. Deshmukh, A. K. M, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*, 2017, pp. 115–124.

[16] C. R. Rocha and R. F. de Mello, "Duplicate bug report detection: A survey," *Journal of Systems and Software*, vol. 145, pp. 104–123, 2018.

[17] A. Budhiraja, K. Dutta, R. Reddy, and M. Shrivastava, "DWEN: Deep word embedding network for duplicate bug report detection in software repositories," in *Proc. 40th Int. Conf. Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 193–194.

[18] F. Rodrigues, F. Bernardino, A. Santos, and R. Novais, "Soft-alignment for duplicate bug report detection using sentence embeddings," in *Proc. 34th Brazilian Symposium on Software Engineering (SBES)*, 2020, pp. 1–10.

[19] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *Proc. 38th Int. Conf. Software Engineering (ICSE)*, 2016, pp. 309–320.

[20] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[21] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *Proc. 19th Working Conf. Reverse Engineering (WCRE)*, 2012, pp. 215–224.

[22] X. Yang, D. Lo, X. Xia, and J. Sun, "Towards automatic bug triage: An adaptive learning approach," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 6, pp. 1398–1410, 2014.

[23] 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.

[24] A. Niculescu-Mizil and R. Caruana, "Predicting good probabilities with supervised learning," in *Proc. 22nd Int. Conf. Machine Learning (ICML)*, 2005, pp. 625–632.

[25] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Chernova, "FAST^2: An intelligent assistant for finding relevant papers," *Expert Systems with Applications*, vol. 120, pp. 57–71, 2019.

[26] E. Kocaguneli, T. Menzies, and J. W. Keung, "On the value of ensemble effort estimation," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1403–1416, 2012.

[27] H. Tu, Z. Yu, and T. Menzies, "Better data labelling with EMBLEM (and how that impacts defect prediction)," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 278–294, 2022.

[28] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "ELBlocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93–106, 2015.

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

[30] C. Tantithamthavorn, J. Jiarpakdee, and A. S. Hata, "The importance of accounting for the impact of data quality and preprocessing in defect prediction," *Empirical Software Engineering*, vol. 26, no. 5, pp. 1–42, 2021.

[31] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on the interpretation of defect models," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 320–331, 2021.

[32] G. K. Rajbahadur, S. Wang, G. A. Oliva, Y. Kamei, and A. E. Hassan, "The impact of feature importance methods on the interpretation of defect classifiers," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2245–2261, 2022.

[33] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you? Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, 2016, pp. 1135–1144.

[34] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017, pp. 4765–4774.

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

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

[37] W. McKinney, "Data structures for statistical computing in Python," in *Proc. 9th Python in Science Conf. (SciPy)*, 2010, pp. 51–56.

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

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2019, pp. 4171–4186.

[40] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2019, pp. 3982–3992.

[41] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics (EMNLP)*, 2020, pp. 1536–1547.

[42] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.