International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

A Hybrid Machine Learning and Retrieval-Augmented Generation Framework for GitHub Issue Classification and Codebase Analysis

Mohd Noman Qadri

Department of Software Engineering, Independent Researcher, India.

Abstract
Modern software engineering faces persistent challenges in issue classification and codebase comprehension at scale. While existing machine learning methods classify software issues, they typically operate in offline batch modes and lack interactive querying capabilities. To address this gap, this paper introduces a hybrid framework that integrates a machine learning pipeline with a Retrieval-Augmented Generation (RAG) prototype. The system utilizes a Term Frequency-Inverse Document Frequency (TF-IDF) vectorizer and a Random Forest classifier trained on a balanced subset derived from 50,000 GitHub issue records. To address the methodological challenge of heuristic label leakage, a leakage-mitigated model was evaluated, achieving a cross-validated F1-score of 0.787 when label-generating keywords were explicitly removed from the training features. Furthermore, a dual-level risk scoring model is hypothesized to assess code quality by blending a file-level prediction with an empirical repository-level commit metric. To facilitate codebase comprehension, a prototype RAG engine utilizing Qdrant vector storage allows for natural-language querying. The prototype implements parsing rules for 14 categories of static code vulnerabilities and code smells. The findings suggest that merging predictive machine learning with generative retrieval mechanisms offers a viable approach for heuristic issue classification, though generalization to genuinely labeled software defects remains for future work.

Keywords: Software issue classification, GitHub metadata mining, Machine learning, Retrieval-Augmented Generation, Code intelligence, Static analysis, Large language models

1. Introduction

As open-source software repositories expand rapidly on platforms like GitHub, the need for automated tools to ensure software quality has become critical. Historically, software teams relied heavily on manual code reviews to identify anomalies. Today, machine learning (ML) systems increasingly enhance these manual processes by identifying bug-prone modules based on historical artifacts [1]. While significant academic progress has been made in heuristic classification, most current systems limit analysis to offline, batch-mode processing. Furthermore, they rarely utilize generative artificial intelligence (AI) to facilitate interactive codebase comprehension.

This paper introduces the GitHub Bug Detection and Codebase Intelligence (GBDCI) platform to address these limitations. The comprehensive system is designed to achieve three primary goals: first, to calculate bug risk scores using hybrid ML; second, to scan commit differences for security flaws and code smells

via a static analysis prototype; and third, to foster conversational interactions with the codebase via a Retrieval-Augmented Generation (RAG) prototype.

The primary contributions of this research are as follows:

- A hybrid issue-classification architecture evaluated on a curated dataset of 50,000 GitHub issues, featuring a rigorous mitigation analysis of heuristic label leakage.
- The integration of a real-time static code analysis prototype designed to identify 14 specified vulnerability, maintainability, and code-smell patterns.
- An architectural prototype for a repository-level RAG interface utilizing dense vector embeddings.
- An open-source implementation of the end-to-end framework to facilitate reproducibility.
- The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 presents the proposed system architecture. Section 4 details the machine learning methodology. Section 5 describes the experimental setup, followed by Section 6 which presents empirical evaluation results. Section 7 discusses the findings, and Section 8 outlines threats to validity. Finally, Section 9 discusses privacy considerations and Section 10 concludes the paper.

2. Related Work

2.1 Machine Learning for Issue Classification

Researchers have actively explored software quality metrics since the 1970s. Early pioneers like Halstead [2] and McCabe [3] proposed complexity metrics to gauge the likelihood of faults based on cyclomatic complexity and code volume. Later studies revealed that machine learning models, when trained on historical commit data, outperform these basic metric-threshold techniques [4]. Ensembles such as Random Forest and gradient boosting frequently serve as strong baselines in standard datasets like the PROMISE suite [5].

2.2 GitHub Repository Mining

Analyzing natural language in issue trackers provides a valuable secondary source of information. Lamkanfi et al. [6] demonstrated that representing bug reports with TF-IDF bag-of-words yields strong accuracy in predicting bug severity. More recently, researchers have applied transformer-based architectures such as CodeBERT [7] and GraphCodeBERT [12] to capture the semantic nuances of programming languages and natural language issue descriptions. While these advanced models attain higher representation accuracy, they demand substantially more computational power for real-time inference, making TF-IDF variants attractive for low-latency web applications.

2.3 Retrieval-Augmented Generation for Code Intelligence

Lewis et al. [8] introduced Retrieval-Augmented Generation (RAG) to anchor large language model (LLM) responses using external document collections. Software engineering applications quickly adopted this approach, leading to tools like CodeT5+ and various intelligent code-search utilities [9, 13]. RAG is commonly utilized in software engineering contexts to inject relevant file chunks directly into the prompt context, bypassing the strict token limits of base models [10].

2.4 Research Gap

The reviewed literature indicates limited integration of these components within a single developer-oriented framework, particularly as research shifts toward LLM-based autonomous systems [14, 15]. Therefore, a gap exists regarding unified platforms that combine NLP-driven issue classification, rule-

based static analysis, and cloud-backed generative codebase querying [16, 17]. Table 1 summarizes this gap relative to existing foundational literature.

Table 1: Literature Comparison on Codebase Analysis Systems

Study	Core Method	Issue Class.	Static Analysis	RAG
Halstead [2]	Code Complexity Metrics	Yes (Formulaic)	No	No
Menzies et al. [4]	Naive Bayes/RF	Yes	No	No
Lamkanfi et al. [6]	Naive Bayes	Yes (Severity)	No	No
Feng et al. [7]	CodeBERT	Yes	No	No
Wang et al. [9]	CodeT5+	No	No	Yes
Guo et al. [12]	GraphCodeBERT	Yes	No	No
Borgeaud et al. [13]	RETRO / General RAG	No	No	Yes
Ours	RF + Gemini RAG	Yes	Yes (Prototype)	Yes (Prototype)

3. Proposed System Architecture

3.1 Overview

The GBDCI platform operates on a robust three-tier architecture utilizing a React 18 frontend, a FastAPI backend (Python 3.10), and a dual-database persistence layer. User data is stored in MongoDB Atlas, while vector embeddings are maintained in Qdrant Cloud. To prevent the application from blocking the event loop during heavy I/O, intensive tasks are executed within asyncio thread pools. Lines of Code (LOC) are estimated by analyzing byte counts from the GitHub Languages API, utilizing principles from the ghloc-web project [11].

3.2 Data Ingestion and Static Analysis

GitHub API Ingestion: The system utilizes PyGithub to authenticate via standard OAuth tokens. To manage GitHub's strict rate limits and pagination boundaries, ingestion is capped and retrieved incrementally, ensuring large enterprise repositories do not overwhelm the application state. The system extracts commit messages, file-level diffs, and natural language issues.

Static Code Scanning Prototype: A custom static analyzer scans each commit difference against 14 regex-based and AST-pattern rules to identify patterns corresponding to vulnerabilities (e.g., SQL injection, eval usage) and code maintainability issues. This serves as an architectural prototype for CI/CD integration.

Risk Scoring Prototype: The predictor component utilizes pre-trained scikit-learn models to evaluate file-level and repository-level risk based on commit history features and community metadata.

3.3 Retrieval-Augmented Generation Engine

The RAG module clones the target repository and filters for standard code extensions. Files are split using a RecursiveCharacterTextSplitter configured with a chunk size of 1500 characters and an overlap of 150 characters to preserve context boundaries without severing function definitions abruptly. Embeddings are generated using the gemini-embedding-001 model (yielding 768-dimensional vectors) and stored in Qdrant Cloud using cosine distance. Natural language queries are processed via gemini-2.5-flash with a temperature of 0.2 to prioritize factual retrieval.

4. Machine Learning Methodology

4.1 Dataset Construction and Heuristic Labeling

The initial NLP issue corpus of 50,000 GitHub records was assembled by downloading CSV files from open-source Kaggle and Embold software bug collections. The datasets were merged and deduplicated based on unique issue IDs. Because raw metadata lacks explicit 'Bug' vs. 'Enhancement' labels, a heuristic approach is employed based solely on the issue title field. The title text was lowercased, and records containing keywords such as 'bug', 'error', 'fail', 'crash', 'exception', or 'issue' were classified as Bugs (class 1), while the remainder were designated as Non-Bugs (class 0).

To prevent bias during training, the dataset was balanced by downsampling the majority class using a fixed random seed. Balancing was performed on the entire corpus prior to cross-validation; while standard for exploratory tasks, this global balancing is noted as a methodological limitation compared to fold-specific downsampling. Table 2 details the dataset breakdown.

Table 2: Dataset Characteristics and Class Distribution

Dataset Phase	Total Records	Bug/Non-Bug Samples
Raw Data	50,000	6,133 Bug / 43,867 Non-Bug
Balanced Subset	12,266	6,133 Bug / 6,133 Non-Bug

4.2 NLP Classification Pipeline and Leakage Mitigation

The classification pipeline is implemented using a scikit-learn Pipeline featuring two primary stages: feature extraction via TfidfVectorizer (5,000 max features, unigrams and bigrams) and classification via RandomForestClassifier (150 trees, minimum samples split of 4, fixed random state 42).

To address the methodological risk of label leakage, two models were evaluated: 1. Naïve Model: Trained directly on the raw text (title and body). Because the TF-IDF vectorizer ingests the exact keywords used to generate the heuristic labels, this model suffers from direct circular leakage. 2. Leakage-Mitigated Model: Prior to vectorization, the label-generating keywords ('bug', 'error', 'fail', 'crash', 'exception', 'issue') were explicitly stripped from the text corpus via regular expressions. This reduces direct keyword overlap and allows the model to evaluate contextual patterns associated with the heuristic issue labels, such as stack traces and null-pointer descriptions.

4.3 Dual-Level Risk Scoring Mechanism

The system calculates a final file risk score Rf by blending the ML probability Pml with an empirical commit heuristic score Hc (derived from commit frequency and lines changed).

$Rf = (0.6 * Pml) + (0.4 * Hc)$

This 60/40 weighting is a design heuristic. It ensures that files with highly anomalous commit sizes or frequent revisions are flagged even if their commit messages lack explicit bug keywords.
For broader context, a separate model evaluates repository-level risk based on community engagement metrics (e.g., star count, pull requests), hypothesizing that low community engagement inversely correlates with long-term unpatched repository risk. However, this repository-level framework is currently an architectural hypothesis awaiting formal experimental validation.

5. Experimental Setup

5.1 Hardware and Software Environment

The experiments were conducted on a local workstation operating Microsoft Windows 11 Home Single Language (64-bit). The system featured an Intel Core processor (Intel64 Family 6 Model 154 Stepping 4 GenuineIntel, 12th Gen Intel Core i7-1255U, 1.7 GHz base clock) and 16 GB of physical RAM. The software environment utilized Python 3.10, scikit-learn 1.5.0, FastAPI, and React 18. A random seed of 42 was strictly enforced across all dataset splits and model initializations to guarantee experimental reproducibility.

5.2 Evaluation Metrics and Cross-Validation

The classification models are evaluated using Accuracy, Precision, Recall, and F1-Score. To ensure statistical stability and reduce split bias, all comparative benchmarking was executed using 5-Fold Stratified Cross-Validation.

6. Results

6.1 Issue Classification Performance

The NLP issue classifier was evaluated using 5-Fold cross-validation on the balanced subset of 12,266 records. Table 3 presents a representative confusion matrix taken from a single test fold (N=2,454) to illustrate absolute prediction counts for the Leakage-Mitigated Random Forest architecture.

Table 3: Confusion Matrix for Leakage-Mitigated Random Forest (Single Fold, N=2454)

	Predicted Bug	Predicted Non-Bug
Actual Bug	995 (TP)	232 (FN)
Actual Non-Bug	286 (FP)	941 (TN)

The confusion matrix in Table 3 represents one of the five validation folds and therefore its metrics are not expected to equal the mean values reported in Table 4.

6.2 Baseline Comparison and Leakage Impact

Table 4 details the performance of the Random Forest models relative to standard machine learning baselines across the 5 folds (values represent the mean ± standard deviation of the 5 folds). 'Naïve' denotes models trained on raw issue titles containing the heuristic label-generating keywords; 'Leakage-Mitigated' denotes the Random Forest trained after the explicit removal of those keywords.
The Naïve Random Forest achieved an F1-Score of 0.9163. However, when the label-generating keywords were removed to mitigate data leakage, the F1-Score stabilized at 0.7874. This leakage-mitigated result provides a less leakage-prone estimate of performance under the heuristic labeling protocol.

Table 4: Baseline ML Comparison (Mean ± Std across 5-Fold Cross-Validation)

Model	Accuracy	Precision	Recall	F1-score
Naive Bayes (Naïve)	0.7894 ± 0.005	0.8352 ± 0.004	0.7212 ± 0.012	0.7739 ± 0.007
Logistic Reg (Naïve)	0.8788 ± 0.005	0.8972 ± 0.008	0.8557 ± 0.003	0.8759 ± 0.005
Linear SVM (Naïve)	0.8762 ± 0.005	0.8875 ± 0.007	0.8617 ± 0.007	0.8744 ± 0.005
Random Forest (Naïve)	0.9104 ± 0.006	0.8601 ± 0.009	0.9804 ± 0.004	0.9163 ± 0.006
RF (Leakage-Mitigated)	0.7825 ± 0.007	0.7700 ± 0.005	0.8056 ± 0.013	0.7874 ± 0.008

6.3 Static Security Scanner Prototype

The static code analyzer parses code to identify 14 primary patterns utilizing regular expressions and AST-based matching. These are explicitly divided into security vulnerabilities and code maintainability smells, as detailed in Table 5. Due to the lack of an independently labeled security dataset, true precision and recall for these static rules cannot currently be mathematically quantified; thus, they serve as a structural prototype for automated code review rather than a fully validated security oracle.

Table 5: Static Scanner Prototype Rule Classification

ID	Pattern Identifier	Classification	Severity
1	eval_usage	Security	Critical
2	sql_injection	Security	Critical
3	hardcoded_password	Security	Critical
4	try_without_catch	Maintainability	High
5	empty_catch	Maintainability	High
6	bare_except	Maintainability	High
7	mutable_default	Maintainability	High
8	null_check	Maintainability	Medium
9	deprecated_api	Maintainability	Medium
10	var_usage	Maintainability	Medium
11	double_equals	Maintainability	Medium
12	console_log	Code Smell	Low
13	todo_fixme	Code Smell	Low
14	magic_numbers	Code Smell	Low

7. Discussion

The naïve Random Forest achieved higher recall and F1-score than the evaluated naïve baseline models; however, its apparent performance is substantially affected by heuristic label leakage. The most significant finding is the impact of the heuristic label leakage: removing explicit bug keywords reduced the F1-Score by approximately 0.13, highlighting that naive TF-IDF models often memorize heuristic tags rather than

learning contextual patterns. However, the leakage-mitigated model retained predictive information associated with the heuristic issue labels, achieving an F1-score of 0.7874. Furthermore, the identification of 14 key patterns during commit ingestion assists developers in catching structural flaws such as arbitrary code execution (eval) and exposed credentials.

8. Threats to Validity and Limitations

Construct Validity: While the leakage-mitigated model removes direct keyword overlap, heuristic labeling inherently limits construct validity. The model classifies issues based on language markers rather than historically verified software defects. Therefore, generalization to genuinely labeled software defects remains a threat until validated on benchmarks like the PROMISE suite. Additionally, the dataset was balanced across the entire corpus before cross-validation; while standard for exploratory tasks, this global balancing is noted as a methodological limitation compared to fold-specific downsampling.

External Validity: GitHub API ingestion is commonly capped by rate limits. This limitation may lower detection recall for older files in extensive repositories. Furthermore, findings derived entirely from GitHub may not perfectly generalize to enterprise environments such as GitLab or Bitbucket, where commit cultures and issue templates differ substantially.

Internal Validity: The weighting schema for file risk (60/40 ML/heuristic split) is an arbitrary design heuristic lacking formal sensitivity analysis. Furthermore, without manually validated ground truth (e.g., the Juliet benchmark), the static analysis module's true precision, recall, and false-positive rate remain unquantifiable. Formal benchmarking of the RAG retrieval accuracy and LLM groundedness is similarly deferred to future research.

9. Ethical and Privacy Considerations

Because the RAG engine relies on cloud-based LLM endpoints (Gemini), proprietary source code retrieved from private repositories must be securely transmitted over network boundaries. The current architecture transmits codebase chunks to the LLM solely for the duration of the generation request. However, this creates an inherent privacy vulnerability. If utilizing external LLM APIs, developers must ensure API configurations explicitly opt-out of data retention policies to prevent sensitive intellectual property or inadvertently hardcoded credentials from being utilized in external model training.

10. Conclusion and Future Work

This paper presented a hybrid platform integrating machine learning, static scanning, and conversational RAG. The system architecture processes commit histories, flags 14 categories of security, maintainability, and code-smell patterns, and creates a dense vector index for generative codebase querying.

Future work will focus on: (1) establishing manually validated datasets to eliminate heuristic labeling completely; (2) performing rigorous ablation and sensitivity analyses on the 60/40 risk score weights; (3) executing comprehensive baseline comparisons with transformer-based embeddings (e.g., CodeBERT) to measure accuracy gains against TF-IDF architectures; and (4) performing formal quantitative evaluation of RAG groundedness using established benchmarks.

Declarations

Funding: This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Conflict of interest: The authors declare that they have no competing interests.
Data availability: The 50,000-record CSV dataset utilized during the current study is available subject to GitHub's standard API terms of service and Kaggle open-source distributions.
Code availability: The source code for the platform is publicly accessible at https://github.com/nomanqadri34/github-bug-detection.
Author contributions: Mohd Noman Qadri is the sole author and assumes responsibility for the conceptualization, methodology, software implementation, validation, and writing of this manuscript.

References

1. T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," IEEE Trans. Softw. Eng., vol. 38, no. 6, pp. 1276-1304, Nov./Dec. 2012.
2. M. H. Halstead, Elements of Software Science. New York, NY: Elsevier, 1977.
3. T. J. McCabe, "A complexity measure," IEEE Trans. Softw. Eng., vol. SE-2, no. 4, pp. 308-320, Dec. 1976.
4. T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Trans. Softw. Eng., vol. 33, no. 1, pp. 2-13, Jan. 2007.
5. P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," Inf. Softw. Technol., vol. 59, pp. 170-190, Mar. 2015.
6. A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in Proc. 7th IEEE Working Conf. Min. Softw. Repos. (MSR), Cape Town, South Africa, 2010, pp. 1-10.
7. F. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in Findings of EMNLP 2020, 2020, pp. 1536-1547.
8. P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2020, pp. 9459-9474.
9. Y. Wang et al., "CodeT5+: Open code large language models for code understanding and generation," in Proc. EMNLP 2023, 2023, pp. 1069-1088.
10. Google DeepMind, "Gemini: A Family of Highly Capable Multimodal Models," arXiv:2312.11805, Dec. 2023.
11. P. Pavlov, "ghloc-web: GitHub Lines of Code Counter," GitHub repository, 2023.
12. D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in Proc. ICLR 2021, 2021.
13. S. Borgeaud et al., "Improving language models by retrieving from trillions of tokens," in Proc. ICML 2022, 2022.
14. X. Yin et al., "A survey on large language models for software engineering," IEEE Trans. Softw. Eng., 2024.
15. C. Gao et al., "Retrieval-Augmented Generation in Software Engineering: A Systematic Mapping," arXiv:2403.02100, 2024.
16. D. Guo et al., "DeepSeek-Coder: When the large language model meets programming," arXiv:2401.14196, 2024.
17. Y. Li et al., "Automated issue classification with large language models," in Proc. ICSE 2023, 2023.