Just-in-Time Bug Classifier: A Step Towards Integrating Automated Program Repair in CI/CD Pipelines

Vinay Kabadi[a,*], Xuan-Bach D. Le[a,**], Patanamon Thongtanunam[a], Christoph Treude[b]

[a] *School of Computing and Information Systems, The University of Melbourne, Australia*
[b] *School of Information Systems, Singapore Management University, Singapore*

Abstract

Context: Automated Program Repair (APR) tools have advanced in recent years, yet their effectiveness can improve when integrated into Continuous Integration/Continuous Deployment (CI/CD) pipelines. Motivated by this, we designed the Continuous Automatic Repair Framework (CARF), which detects build failures, routes each bug to the most suitable repair tool, and automatically commits the generated fix. During development, we identified a critical bottleneck: the need for instant bug classification within CI/CD. Accurate classification is essential to determine whether a fault lies in program code or the test suite, ensuring the defect is routed to the appropriate repair tool.

Objectives: This study aims to design and evaluate a just-in-time bug classifier capable of distinguishing between program and test bugs during CI/CD execution, thereby enabling CARF to maintain workflow efficiency by directing defects to appropriate repair tools.

Methods: We implemented a heuristic analysis tool that extracts discriminative structural features by comparing Abstract Syntax Trees (ASTs) between buggy and pre-buggy commits. These features are input to machine learning models for bug classification. Empirical validation demonstrated the

*Corresponding author: Vinay Kabadi
**Corresponding author: Xuan-Bach D. Le
Email addresses: vkabadi@student.unimelb.edu.au (Vinay Kabadi), bach.le@unimelb.edu.au (Xuan-Bach D. Le), patanamon.t@unimelb.edu.au (Patanamon Thongtanunam), ctreude@smu.edu.sg (Christoph Treude)

accuracy and operational feasibility of the approach within CI/CD environments.

Results: Our approach achieved up to 73% accuracy in identifying regression bugs across 67 real-world projects, effectively distinguishing between program bugs and test bugs while requiring only 10% of the dataset for training. In contrast, prior studies reported an accuracy of 69% on artificially injected bugs derived from successive versions of only two projects, with 90% of the data used for training.

Conclusion: CARF facilitates the integration of automated program repair into CI/CD pipelines, enabling faster, more accurate, and more flexible software maintenance. The just-in-time bug classifier demonstrates that defect classification can scale efficiently without symbolic execution or manual bug reports, providing a practical foundation for continuous automated repair.

Keywords:
Automatic Program Repair, Bug Classifier

---

1. Introduction

In 2020, operational software failures cost the U.S. economy $1.56 trillion, of which software debugging, testing, and verification costs account for more than $100 billion annually, which is up to three-quarters of the total software development budgets and taking up to half of software engineers' time [1]. In projects with long life-cycles, it becomes crucial to regularly maintain and update both the program and the test suite to prevent code breakage from these sources. They both play significant roles in causing software issues and also contribute differently to code-breaking scenarios.

Fortunately, the **Automatic Program Repair (APR)** research space is rapidly evolving, developing tools that can automatically fix software bugs and test cases. Though APR has shown significant impact and has been deployed by a few IT giants, it is still far from being omnipotent and faces shortcomings that hinder its widespread adoption. Kabadi et al. [2] have demonstrated that most automated program repair (APR) tools are evaluated in curated environments rather than in real-world regression bugs, raising concerns about the credibility of their claimed effectiveness and also the risk for having introduced some bias [3]. To achieve broader implementation, an effective strategy would be to continuously test the APR tools

with regression bugs by incorporating them into continuous integration (CI) pipelines.

In a typical CI/CD pipeline, a GitHub commit triggers a series of automated processes. First, the code is compiled to ensure it can be transformed into executable software. This step verifies the syntax and structure of the code to detect any errors early in the development cycle. Subsequently, automated tests are executed to validate the software's functionality and performance. These tests include unit tests, integration tests, and possibly end-to-end tests, aimed at maintaining the stability and reliability of the codebase throughout its development. Finally, upon successful testing, the software is deployed to a staging environment or directly to production, depending on the pipeline configuration.

Problems arise when a build fails in the CI/CD pipeline. For those with failed test cases, the issues are generally due to either program bugs or test bugs. Program bugs typically result from defects or unintended behaviours in the source code, such as logic errors or unexpected interactions among software components. However, failing tests are not always indicative of code defects. In many cases, outdated or obsolete test cases pose a significant challenge, especially in rapidly evolving software environments [4]. As the codebase evolves, test cases may become misaligned with the system's intended behaviour, leading to false positives (tests that fail despite the code being correct) or false negatives (tests that pass despite the presence of actual bugs). Such inconsistencies can obscure real defects or mistakenly flag correct implementations, thereby misguiding debugging efforts and delaying issue resolution. Over time, the inability of test cases to evolve alongside the system reduces their reliability. This makes it crucial to distinguish between failures caused by code defects and those caused by faulty or outdated tests. Supporting this, an internal study of a product at ABB [22] found that 80% of post-release defects over a three-year period were either latent bugs introduced in earlier releases or regression defects caused by subsequent code changes, further highlighting the complexity of accurately attributing failure causes.

It is worth noting that techniques for automated program repair typically assume that test cases are correct and that the fault lies within the program code. Similarly, automated test repair techniques often assume that failures are due to obsolete or incorrect tests [5]. However, these assumptions do not always hold true in practice. In many real-world scenarios, the actual cause of failure may not align with these expectations, such misclassifications

can introduce significant bias and inefficiencies in bug triage and automated repair processes. Especially in CI/CD pipelines, where instant bug classification becomes critical to maintain development velocity and reliability. Kim et al. [6] observed that nearly one in every three reported bugs is not an actual bug, highlighting the prevalence of misclassifications. This makes it crucial to first identify whether the failure stems from the code or the tests before selecting an appropriate repair strategy [5].

We observed that a wide range of automatic repair tools have been developed to address different types of software issues, which could potentially resolve the challenges described above. However, to fully leverage their capabilities, it is essential to integrate these tools into a framework that ensures the right tool is applied to the right problem. Motivated by this need, we envisioned the Continuous Automatic Repair Framework (CARF), as illustrated in Fig. 1, which integrates several state-of-the-art techniques to enable effective automatic repair within the CI/CD pipeline.

The Continuous Automatic Repair Framework (CARF) is an intelligent automation layer within the CI/CD pipeline. Upon a build failure, CARF invokes an instant bug classifier to determine whether the fault resides in the program code or the test suite. It then leverages tools such as the E-APR framework [7] to select and rank the most suitable repair strategies for the specific software issue. Before applying the selected tool, CARF performs a feasibility check [8] to ensure that the tool can generate a valid patch. If a valid patch is produced, it is committed back to the repository; otherwise, the issue is deferred for manual resolution. By automating failure triage and repair, CARF enhances both reliability and agility in CI/CD pipelines without introducing additional risk.

However, while most other aspects could be adapted from state-of-the-art work, a crucial gap remained for the effective implementation of CARF. This gap is the need for an instant bug classifier that distinguishes between program and test bugs with high accuracy. Most bug classification tools rely on bug reports or symbolic/runtime execution, but these methods are unsuitable for just-in-time classification. Bug reports are manually created long after the bug appears, and runtime execution requires setting up environments, which is impractical for every failing bug in a CI/CD pipeline. To address these challenges, we developed a faster and more reliable solution that eliminates delays caused by waiting for bug reports or configuring execution environments. Our method relies solely on static code of both buggy and previous versions, removing external dependencies and enabling

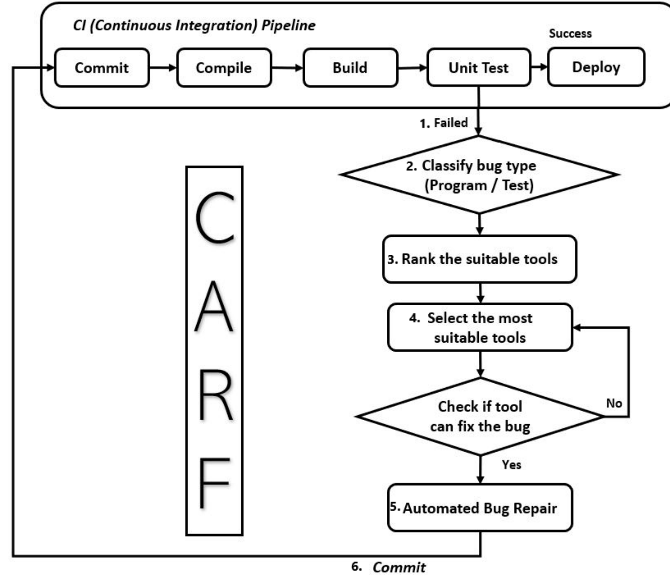

Figure 1: Continuous Automated Repair Framework (CARF).

just-in-time classification directly within the CI/CD pipeline. This makes it well-suited for fast-paced CI/CD workflows where speed and immediate response are crucial.

Thus, we have developed a just-in-time bug classification model using a dataset of regression bugs. The artifact is available on GitHub[1] for both the bug classifier and the dataset. To evaluate its effectiveness, we explore the following two research questions:

RQ1 *How effective are different machine learning models in classifying regression failures as program or test bugs across diverse projects?* To address this research question, we chose to utilise Support Vector Machine (SVM) models as they are known for providing robust classification even with limited training data and have demonstrated strong performance with Boolean models. We conducted a comparative analysis of various SVM models, including ensemble and neural networks, to evaluate their classification accuracy and efficiency in distinguishing between program and test bugs.

[1] `https://github.com/CI-Bugs/bug_classifier`

RQ2 *Which structural features of code changes (derived from AST differencing) contribute most to distinguish program from test bugs?* To answer this research question, we have extracted the abstract syntax tree (AST) structure from the differential code between the buggy commit and the pre-buggy commit. By analysing these AST features, we identify the most distinguishing features that classify the program and test bugs.

In summary, the contributions of this work are as follows:

- Continuous Automatic Repair Framework: A conceptualised Continuous Automatic Repair Framework (CARF), illustrated in Fig. 1, has been designed to automate repair within CI/CD pipelines. A just-in-time bug classifier was identified as a critical gap in implementing it.

- Bug Classification Model: We developed a machine learning-based classification model that requires only the URL of the buggy commit as input. The model automatically retrieves both buggy and pre-buggy code versions, extracts features using Abstract Syntax Tree (AST) analysis and static code differencing, and classifies the bug as either program or test related. By eliminating the need for manual inspection, external dependencies, or runtime execution, this model enables just-in-time bug classification within the CI/CD pipeline, facilitating faster and efficient automated repair.

- Dataset of Regression Bugs: A curated dataset of regression bugs (derived from the Bugsswarm dataset) was created to simulate just-in-time scenarios, providing a practical foundation for testing bug classification in continuous integration environments.

- Prominent List of Code Features: A set of features was identified and extracted from the static code differences between buggy and pre-buggy commits, significantly aiding in distinguishing program bugs from test bugs.

2. Motivation

Automated repair tools have evolved significantly across multiple domains; however, they often operate in isolation without integration into a

unified ecosystem that leverages just-in-time feedback from live development environments, such as CI/CD pipelines. Integrating these tools to work collaboratively and continuously learning from ongoing bugs, such as regression failures, can improve their adaptability and effectiveness in real-world scenarios.

Herzig et al., [6], examine how misclassifications of bugs and features introduce biases into bug prediction models. By manually analysing over 7,000 issue reports from five open-source projects, the authors found that 33.8% were misclassified, leading to confusion between actual bugs and features. These misclassifications significantly affect the accuracy of bug prediction and introduce bias in bug prediction models.

This realisation led us to conceptualise the Continuous Automatic Repair Framework (CARF), an orchestration layer that detects build failures, classifies bugs, and invokes suitable repair tools with minimal human intervention. While designing CARF, we identified a critical missing piece: an instant, just-in-time bug classifier capable of distinguishing between program and test bugs, essential to avoid misrouting and ensure effective repairs.

The motivation behind our work was to build the Continuous Automatic Repair Framework (CARF). However, the absence of an instant, just-in-time bug classifier emerged as a critical blocker. This gap directly motivated us to develop a machine learning based bug classification model that enables just-in-time categorisation of bugs, an essential component for the successful implementation of CARF. An additional advantage of using machine learning in this context is that all the tools we aim to integrate within the framework are also based on ML models, making it feasible to incorporate them into a unified, compatible ecosystem.

Hence, this research is motivated by the need for high-level semantic bug classification, specifically distinguishing between program and test bugs, which is critical for implementing CARF. Existing manual or dynamic classification methods are either too slow or require complex runtime environments, making them impractical for fast-paced CI/CD workflows. In contrast, static code comparison, especially at the Abstract Syntax Tree (AST) level, enables instant identification of bug causes and change types. Building on this insight, we developed an automated, machine-learning-based bug classification approach designed for just-in-time integration into CI/CD pipelines, enabling precise repair tool selection and continuous, context-aware software repair.

3. Background and Related Work

Bug classification in software engineering is generally approached in three main ways. The first and most common approach is based on bug reports from issue tracking systems. Using techniques such as Natural Language Processing (NLP), bugs can be categorised according to their descriptions, keywords, severity, and recurrence. The second approach relies on symbolic execution, which developers primarily use for real-time bug detection and fixing. This method requires a fully executable environment with all binaries and dependencies, allowing semantic analysis of the code by exploring different execution paths using symbolic inputs. The third approach, and the focus of our research, is analysing the static code. This involves examining the source code without execution to identify potential issues and classify bugs based on changes or patterns that may lead to failures.

Existing work on static bug classification based on prior failures has been limited. The studies have only been validated on two projects with manually injected bugs and have not been thoroughly tested on real-world regression bugs [5]. Additionally, the datasets used in these studies have not been publicly released, limiting their reproducibility and broader applicability. These gaps highlight the need for a more robust, just-in-time bug classification system that can accommodate the dynamic nature of continuous development environments. This also highlights the need for approaches like ours that integrate just-in-time classification directly into CI/CD workflows.

The closest related work to our approach is the study "Is This a Bug or an Obsolete Test" [5], which compares static code between successive versions of projects to determine whether a test failure is caused by an actual bug or by outdated test cases. Despite training and validating on the same projects with manually injected bugs, the study achieved an accuracy rate of 69%. This highlights the challenges faced in static bug classification, where Dan et al. [5] use the Best-first Decision Tree Learning algorithm to analyse data from test failures. To evaluate this approach, three studies were conducted on two Java programs, Jfreechart and Freecol:

1. Same Version Study: Assessed effectiveness when training and testing on the same version of a program.
2. Different Versions Study: Tested effectiveness across different versions of a program.
3. Different Programs Study: Evaluated the effectiveness of different programs.

The shortcomings of this research are:

1. The dataset is not available for further investigation.
2. The faults are manually injected into the product code, hence they do not represent real regression bugs.
3. The training is conducted on a very limited dataset, which is actually different versions of the two projects, raising concerns about bias.
4. The accuracy rate is only 69%, despite training and testing on the same dataset, where 90% of the data is used for training and 10% is reserved for testing.
5. The machine learning model is trained and tested on the same projects and may therefore be effective only for those specific projects, without guarantees of general applicability to other projects.

Although bug report-based classification has been widely studied and achieved notable success, it is less suitable for real-time CI/CD environments because bug reports are often generated manually and only after failures occur. This latency makes them incompatible with inherently continuous CI/CD pipelines.

Foundational efforts like IBM's Orthogonal Defect Classification (ODC)[9] laid the groundwork for modern bug classification techniques, numerous program and test repair tools[10] have shown effectiveness in specific contexts.

Automated Program Repair (APR) evaluations have traditionally focused on aggregate success rates, which provide limited insight into why particular repair techniques succeed or fail on different kinds of bugs. Aleti et al address this gap in E-APR[7] by proposing an instance-space analysis that represents each buggy program as a point in a high-dimensional feature space. The features capture structural, behavioural, and test-execution characteristics (e.g., code complexity, test coverage, behavioural variation), and dimensionality-reduction techniques (such as PCA) are used to produce interpretable visualisations. These visual landscapes reveal regions of repair instances and enable analysis of the diversity and quality of bug instances used in empirical APR studies, rather than relying solely on coarse success metrics. By projecting the historical success and failure outcomes of different APR systems onto this instance space, E-APR enables evidence-based tool selection. A new bug can be characterised with the same features and located within the instance space to identify which repair approaches have historically performed well in that region. Thus, E-APR transforms APR evaluation into a

feature-aware diagnostic process that reduces trial-and-error, guides practitioners to the most promising repair tools for a given instance, and highlights sparsely covered or challenging regions that warrant future research and the development of tailored techniques.

A practical application of Automated Program Repair (APR) is hindered by significant, unpredictable computational costs, as many bugs cannot be fixed within a reasonable time frame, resulting in wasted developer resources and increased wait times. To address this inefficiency, Le et al. [8] proposed a technique centred on the delegation decision, determining early in the process whether a bug fix should be delegated to an APR tool or reverted to manual debugging. Their core contribution is a machine learning oracle that predicts the effectiveness of a Search-Based APR (SAPR) technique within the first few seconds of its execution. This oracle operates by extracting critical features, including the program size, the nature of test cases, the locality of suspicious code, and the diversity of initial repair candidates, to train a discriminative model. By classifying a repair instance as either effective (likely to succeed within a time budget) or ineffective (unlikely to succeed), the system prevents engaging in time-consuming, unproductive attempts, thereby making the overall use of automated repair more practical and cost-effective.

Orthogonal Defect Classification (ODC), proposed by Chillarege et al. [9], provides a structured methodology for classifying defects by their root causes and the stage of the software development lifecycle at which they are detected, enabling in-process feedback to developers. Building upon ODC, AutoODC [11] eliminates the need for manual intervention by automating this process using machine learning to generate ODC labels for software defects. This automation significantly reduces the time and effort for defect classification, enhancing ODC's scalability in large projects and improving the consistency and accuracy of defect labelling, leading to more reliable defect prediction models.

Automatic Defect Categorisation [12] by Thung et al. introduces a method to automatically categorise bug reports into three families: control and data flow, structural, and non-functional. This approach leverages machine learning and natural language processing (NLP) to classify defects based on their textual content, streamlining the process. By extracting features from defect descriptions, the system efficiently categorises defects, facilitating faster resolution and improved resource allocation. The use of NLP in this context has shown promising results, improving the accuracy of bug-prediction models

by up to 77% through semantic understanding of defect reports.

Xia et al. [13] propose a novel defect classification approach that analyses fault triggering conditions rather than relying solely on defect descriptions. This method offers more precise categorisation, especially when descriptions are incomplete or ambiguous. Understanding the underlying causes improves defect classification and offers insights for future prevention.

Active Semi-Supervised Defect Categorisation [14] by Thung et al. combines labelled and unlabelled data using active learning. It reduces labelling effort while maintaining accuracy, making it suitable for real-world software projects. Automatic Bug Triage using Semi-Supervised Text Classification [15] presents a model that assigns bug reports to developers using bug descriptions. It improves triage accuracy in large projects by reducing manual overhead. Similarly, Neelofar et al. [16] introduce a machine learning solution for classifying software bugs. It analyses bug reports, using textual content, historical data, and metadata to categorise bugs (e.g., functional, performance, security). This automates classification, reducing manual effort and improving consistency, with emphasis on feature selection for accuracy, ultimately ensuring efficient defect resolution.

González et al. [17] introduce a crowd-sourced approach to defect classification. By leveraging insights from diverse contributors, their method enhances model accuracy and robustness, effectively overcoming the limitations of traditional methods through broader perspectives and expertise.

Nagwani et al. [18] present the LDA (latent Dirichlet allocation) method for automatically generating a hierarchical taxonomy of bug types by analysing software bug reports. It helps in identifying latent themes and patterns in bug data, facilitating the creation of a comprehensive taxonomy. This classification framework aims to improve the organisation and retrieval of bug-related information, thereby enabling more efficient bug management and resolution.

Leszak et al. [19] explore methods for classifying and evaluating defects after project completion. The study introduces a classification framework to categorise defects into functional, performance, and usability issues. It also defines evaluation metrics such as defect density and time-to-resolution. By analysing historical defect data, the paper provides insights into defect patterns and root causes, offering recommendations for continuous improvement in defect management.

Denger et al. [20] present a case study on the implementation and validation of a defect classification system in an industrial setting. The paper discusses how defect classification was integrated into process improvement

and quality management practices. The study highlights the benefits of structured defect classification, including identifying quality issues, streamlining defect-resolution processes, and enhancing overall product quality. The case study provides practical insights into the challenges and successes of applying defect classification in real-world scenarios.

Xuan et al. [21] address the challenge of bug triage by employing software data reduction techniques. The study proposes methods to reduce the volume of bug reports by filtering and summarising relevant data, which facilitates more effective triage and prioritisation. The research highlights the importance of mitigating data overload to enhance the efficiency of bug management processes. The proposed techniques aim to streamline the triage process, enabling quicker identification and resolution of critical bugs.

Antoniol et al, [22] explain how linguistic information or the textual content in bug tracking systems aids in automatically distinguishing the change requests into corrective maintenance and other kinds of activities. Machine learning classifiers are trained on these features to automate the classification process with up to 82% accuracy;

4. Continuous Automated Repair Framework

The Continuous Automated Repair Framework (CARF) is designed to integrate Automated Program Repair (APR) seamlessly into existing CI/CD pipelines [23], thereby reducing manual intervention and improving software quality. Figure 1 illustrates the overall architecture of CARF, which consists of six core components: (1) detection of failed commits, (2) classification of bug type, (3) ranking of suitable APR tools, (4) feasibility checking of tool, (5) automated bug repair, and (6) recommitment of fixes. Each component operates in a tightly coupled sequence to ensure that failures are diagnosed and addressed in real time, without disrupting the normal flow of continuous integration or deployment. The proper functioning of the framework is dependent on the inputs and outputs received and sent by each module. The primary inputs to this framework are the buggy versions of the code and the intended APR tools that will be configured and set up for execution, the overall success is critically dependent on the characteristics and operational behaviour of these components.

1. **Detection of failed commits:** Whenever code changes are pushed to Git, the CI/CD pipeline is triggered to compile, build, test, and deploy

the software under ideal conditions. CARF continuously monitors these builds and automatically detects any failures. As soon as a test failure is reported, CARF flags the failed commit and initiates the next stages of CARF to diagnose and resolve the issue.

2. **Classification of bug type:** Once a failure is detected, CARF triggers its just-in-time classifier, comparing the current and previous commits to determine whether the root cause is a program bug (an error in application code) or a test bug (an error within the test case itself). This distinction is critical to ensure that only repair tools relevant to that bug category are considered.
3. **Ranking suitable APR tools:** After identifying the bug category, CARF queries a tool selector such as E-APR [7] to rank the deployed APR tools by relevance. To ensure the selection of the most effective APR tool for a given issue, the framework uses inputs that include the buggy program version, the corresponding failing test suite, the associated failing test cases, and the deployed APR tools. The E-APR framework addresses this by first constructing performance footprints for each APR tool based on their performance results, which represent each technique's respective areas of strength. It then employs a genetic algorithm to identify the most discriminative features from the buggy program instances that effectively distinguish between the scenarios. Using these optimised features, E-APR generates a two-dimensional instance space that visualises the relationships between problem instances, their characteristics, and the observed performance of APR techniques. Finally, E-APR applies machine learning methods to the most significant features to learn a predictive model that recommends the most suitable APR tool for future repair tasks.
4. **Feasibility check for a suitable tool:** Before invoking the selected repair tool, an additional optimisation layer is introduced to assess the likelihood that the tool will successfully resolve the bug. Allowing an unverified tool to attempt a repair can be costly, as some tools may fail to generate a valid patch or require excessive time, potentially delaying the CI/CD pipeline. Building on this idea, oracle tools are deployed, such as those developed by Le et al. [8]. These tools extract the set of features from the faulty program and its associated test cases, and then test them with the APR tools to assess the suitability of the recommended tool. The approach leverages dynamic features gathered during a brief initial execution period (30 seconds) of the Search Based

Automated Program Repair (SAPR) technique. These features include the locality of suspicious program points identified by fault localisation and the diversity of the initial population of repair candidates. This feature set is then used as input to a machine learning algorithm, which builds a discriminative model to predict the repair attempt's effectiveness. If the tool is predicted to be ineffective, the system automatically proceeds to the next most appropriate tool suggested by E-APR, repeating this process until a suitable repair tool is identified or it finally fails, allowing manual intervention, ensuring a more efficient overall bug-fixing process.

5. **Automated Bug Repair:** The selected APR tool is then triggered to repair the bug. For program bugs, the tool analyses the faulty code, generates candidate patches, and tests them to ensure the issue is resolved without introducing regressions. For test bugs, it inspects the test cases, applies the necessary corrections, and re-validates them. If the bug remains unresolved, the process reverts to manual intervention, as it would in the absence of CARF.
6. **Recommit:** When a valid patch is produced, CARF automatically commits the fix back into the version control system and re-triggers the CI/CD pipeline. The build is then re-triggered, allowing the process to continue by rerunning the tests to confirm the software's integrity. Importantly, if CARF fails to generate a valid patch, the process simply falls back to manual intervention just as it would have without CARF. Thus, CARF introduces no downside or regression risk; it only enhances the existing process by providing an opportunity for automated recovery where possible.

5. Bug Classifier - Methodology

In this research, we focus on developing an instant, just-in-time bug classification method that addresses the critical gap identified earlier, enabling the effective implementation of CARF. Our approach leverages static code analysis, a foundational software engineering technique that detects bugs without executing the program. This non-intrusive method is well-suited for instant bug classification, as it avoids reliance on runtime environments or external dependencies. Instead, it analyses readily available structural and historical changes in the source code to identify potential issues early in the

development cycle, making it an ideal technique for proactive bug detection and classification.

Specifically, we analyse Abstract Syntax Trees (ASTs) generated from parent and child commits within a GitHub repository. ASTs provide a hierarchical tree representation of the source code's syntactic structure, capturing relationships among functions, variables, control flow, and other elements. By comparing ASTs from buggy and pre-buggy commits, we identify and highlight structural code changes that may introduce bugs. This comparison enables us to classify bugs effectively by revealing patterns that distinguish program-related defects from test-related failures.

Similar methods have been employed by other researchers for various objectives. For example, Bermejo et al.[24] investigated bug-introducing changes (BIC) and bug-fixing changes (BFC) to identify the exact code edits responsible for defects. Zhen Ni et al.[25] examined code elements and structural patterns in bug fixes by constructing "fix trees" from AST-level diffs, leveraging the relationship between bug causes and their corresponding fixes to automatically classify bugs into cause categories. Additionally, Zhao et al. [26] analysed change types in bug-fixing code, categorising them into five change types and nine subtypes. These studies illustrate that analysing static code differences at the AST level provides valuable insights for both understanding and classifying bugs.

To classify bugs effectively, we followed a structured methodology comprising three key steps:

Data Collection: We sourced our data from the Bugswarm [27] dataset, a repository of known bugs that provides Fail-Pass pairs along with their associated commit histories. For each bug, we collected the commit links from Bugswarm and retrieved the commit history from the corresponding GitHub repositories. Our focus was on identifying buggy commits and their immediate predecessor commits (pre-buggy commits) to establish a direct comparison between the code that caused the bug and the state of the code prior to the bug.

AST Generation: Once the relevant commits were identified and the repositories were downloaded, we generated Abstract Syntax Trees (ASTs) for both the buggy commit and the pre-buggy commit. This was accomplished using the Gumtree [28] tool, a sophisticated software designed to perform detailed tree-based comparisons of code. Gumtree generates ASTs and identifies differences between two versions of code by analysing their structure rather than relying on line-by-line comparisons. This made it well-

suited for our analysis, as it allowed us to focus on the syntactic and semantic changes in the code.

AST Comparison for Bug Classification: After generating the ASTs, we conducted a comparison between the AST of each buggy commit and the AST of the corresponding pre-buggy commit. By identifying the structural differences between the two code versions, the specific changes that could have introduced the bug can be identified. These differences are essential for classifying the nature of the bug – whether it originated from program logic, test code, or build configuration changes. By examining elements such as added, removed, or modified nodes in the AST, we extract a set of features that highlight key structural and functional modifications. These features serve as the basis for classifying the bugs.

To perform the comparative analysis, we examine differences at the Abstract Syntax Tree (AST) level to determine whether code changes align with software development assumptions. Our approach is guided by fundamental intuitions from software debugging, which can indicate whether a failure originates from a program bug or a test bug.

1. **Deleted Program File / Class / Function** (CFD / CFD-CD / CFD-CD-FD / CFM-CD / CFM-CD-FD / CFM-CM-FD): If a test fails after a program class or function is deleted and if the corresponding test case is not updated, it is likely a test bug. The feature was removed, but the test case continued to expect the deleted functionality, resulting in a failure.
2. **Added Program File / Class / Function** (CFA / CFA-CA / CFA-CA-FA / CFM-CA / CFM-CA-FA / CFM-CM-FA): If a bug appears after a new class or function is added, but no new or updated test cases are introduced, it is likely an indication of a program bug. The new code may have introduced unintended side effects, causing existing tests to fail.
3. **Added Program File / Class / Function with Updated Test Cases** (CFA-CA-FA-TU / CFM-CM-FA-TU / CFM-CM-FM-TU): If a bug appears after adding a new class or function and updating the corresponding test cases, it likely indicates a program bug. The updated test cases correctly target the new functionality, and their failures suggest an issue with the implementation.
4. **Modified Program File / Class / Function** (CFM / CFM-CM / CFM-CM-FM): If a test fails after a function or class is modified

without corresponding updates to test cases, it likely indicates a test bug. The functionality was changed, but the test case still reflects outdated expectations.

5. **Added Test File / Class / Function** (TFA / TFA-CA / TFA-CA-FA / TFM-CA / TFM-CA-FA / TFM-CM-FA): If a test fails after a new test file, class, or function is added, the issue likely lies in the program. This suggests that the corresponding functionality is not yet implemented or is incomplete, resulting in the new tests failing.
6. **Modified Test File / Class / Function** (TFM / TFM-CM / TFM-CM-FM): If a test fails after a test class or function is modified, the issue is likely to lie within the program. The updated test may now expose unimplemented or incorrect behaviour in the underlying program code.
7. **Deleted Test File / Class / Function** (TFD / TFD-CD / TFD-CD-FD / TFM-CD / TFM-CD-FD / TFM-CM-FD): If a test fails after a test class or function is deleted while the program remains unchanged, the system's behaviour becomes ambiguous. Such deletions may not align with the program's current state, resulting in unpredictable outcomes.
8. **Non-Code File Changes** (NCF): Changes in non-code files (e.g., configuration files, documentation, or README files) were excluded from the analysis. These files provided insufficient context for determining the origin of the fault and were therefore treated as noise in later stages of the research.

This methodology, grounded in analysis of static code, ensures that our classification process remains independent of runtime factors and execution environments. By focusing on syntactic changes in the structural representation of code, we efficiently classify bugs while aligning with real-world software development practices, where early detection and prevention are critical to software quality.

6. Experimental Setup

Given that our proposed tool is intended for use in just-in-time CI/CD environments, we curated a dataset of regression bugs with labels for program and test bugs, ensuring reliable, reproducible results in the practical evaluation setting. In contrast, prior work, such as [5], has used synthetic or

manually injected faults. Such faults often do not capture the complexity of real-world scenarios.

We leveraged the Bugswarm dataset [27], a large-scale collection of reproducible regression bugs comprising over 2,000 builds across Java and Python projects (we used only Java projects in this research). Bugswarm supports widely used build systems, such as Maven, Ant, and Gradle, and is distinguished by its provision of Fail-Pass commit pairs, systematically labelled into categories: Code, Test, and Build failures. The creators of Bugswarm employed multiple iterations of build reproduction to ensure the reproducibility and reliability of each bug instance, making it a robust and trustworthy foundation for empirical research. Given the rigour of its construction and repeated validation for consistency, we utilise this dataset at face value for our analysis. Each bug instance is accompanied by a comprehensive JSON file containing metadata such as the diff URL, buggy and fixed commit hashes, programming language, branch details, build system, bug classification, and build stability.

For this experiment, we focus exclusively on Java bugs for several reasons: (i) previous comparable studies were conducted on Java, enabling reliable benchmarking and result comparison, (ii) the Bugswarm dataset contains more Java bugs than Python, providing a richer experimental base for Java, and (iii) restricting the study to a single language helps avoid potential language-related bias in the results. At the time of data extraction, Bugswarm contained 654 unique Java builds with fail-pass pairs from 67 diverse projects, of which 326 represented program bugs and 244 represented test bugs. However, when extracting specific features, only 288 program bugs and 211 test bugs meet the criteria that have at least one feature from our list (refer github repo[2])

To realise the Continuous Automatic Repair Framework (CARF) as described, the first essential step is to classify bugs into specific types that align with the capabilities of available APR tools. To gain insight into existing APR tools and to establish a high-level bug categorisation, we refer to the living review by Monperrus et al. [10]. This review highlights a range of tools that can be broadly classified into three main categories:

1. Program repair tools: To repair bugs related to dynamic and static errors in the code.

[2] `https://github.com/CI-Bugs/bug_classifier`

2. Test repair tools: To repair or delete obsolete test cases that are no longer required or need to be updated.
3. Domain-specific repair tools: To fix issues related to build scripts, web applications, security, integration, performance, UI, databases, and hardware, among others.

We have scoped our research to program and test bugs only, and we are not venturing into domain-specific bugs, as they encompass a wider range of issues beyond the scope of this research.

Thus, in this research, we focus on building a just-in-time bug classifier that analyses the static code by extracting the AST and categorizes bugs into two groups: program bugs and test bugs. To perform this classification, we analyse the buggy commit by comparing it with its parent commit, which helps us identify the code changes and the nature of the change that caused the failure. This design mirrors real development scenarios, where developers diagnose and triage failures using only the information available at the moment the build fails.

After identifying the buggy and pre-buggy commits, we analyze the differences between the two code states. For this, we used Gumtree [28], a specialized tool that generates diffs based on Abstract Syntax Trees (ASTs) rather than traditional line-by-line comparisons. By comparing the code's structural elements using AST differencing, we extracted 34 distinct features that capture the structural and functional changes introduced by the buggy commit. These features encapsulate critical aspects of the code modifications, including the type and location of code added, removed, or altered, as well as changes in control flow and variable usage. This detailed, syntax and semantics-aware analysis provides a comprehensive view of the commit's impact on the codebase and serves as a foundational input to our bug classification process.

Code-related features:

CFD: An existing code file is deleted.

CFD-CD: A class is deleted in the existing program file.

CFD-CD-FD: A function is deleted in an existing class in the existing program file.

CFA: A new program file is added.

CFA-CA: A new class is added in the new program file.

CFA-CA-FA: A new function is added in a new class in the new program file.

CFA-CA-FA-TU: A new function is added in a new class in the new program file and the relevant test case is updated.

CFM: An existing program file is modified.

CFM-CD: A class is deleted in the existing program file.

CFM-CD-FD: A function is deleted in an existing class of an existing program file.

CFM-CA: A new class is added in the existing program file.

CFM-CA-FA: A new function is added in a new class in the existing program file.

CFM-CA-FA-TU: The relevant test case is updated.

CFM-CM: A class has been modified in an existing program file.

CFM-CM-FD: A function is deleted in an existing class in the existing program file.

CFM-CM-FA: A new function is added in the existing class of an existing program file.

CFM-CM-FA-TU: The relevant test case is updated.

CFM-CM-FM: A function has been modified in an existing class of an existing program file.

CFM-CM-FM-TU: The relevant test case is updated.

Test-related features:

TFD: Existing test file is deleted.

TFD-CD: A test class is deleted in the existing test file.

TFD-CD-FD: A function is deleted in a class in the existing test file.

TFA: A new test file is added.

TFA-CA: A class is added in the new test file.

TFA-CA-FA: A function is added in a new class in the new test file.

TFM: An existing test file is modified.

TFM-CD: A class is deleted in the existing test file.

TFM-CD-FD: A function is deleted in an existing class of an existing test file.

TFM-CA: A new class has been added to the existing test file.

TFM-CA-FA: A new function has been added in an existing class of an existing test file.

TFM-CM: A class has been modified in an existing test file.

TFM-CM-FD: A class has been modified by deleting a function in an existing class in an existing test file.

TFM-CM-FA: A class has been modified by adding a function in an existing class in an existing test file.

TFM-CM-FM: A function has been modified in an existing class of an existing test file.

NCF: Number of non-code files.

However, due to the limited size of our dataset, we faced constraints on the analytical approaches we could employ. While large language models (LLMs) are highly effective for analysing code at scale, their training requires significantly larger volumes of data to perform well. Training an LLM on a small dataset like ours would yield suboptimal performance because the model relies on extensive training data to capture nuanced relationships and patterns in the code.

Given these limitations, we opted to use a Support Vector Machine along with ensemble and neural network approaches for our classification task. SVMs are particularly well-suited for working with small datasets, as they can perform classification even with limited data. By focusing on maximising the margin between different classes (in our case, program bugs and test bugs), SVMs can effectively generalise from a smaller feature set. This made SVM a logical choice for our analysis, as it is known to deliver reliable performance in scenarios where large-scale data is not available. All machine learning models used in this research were implemented using scikit-learn in a Python 3 environment, and the hyperparameters are implemented in the code repository.

In summary, we used Gumtree to extract meaningful features from structural differences between commits and applied SVMs to classify them. This

combination enabled us to tailor our approach to the limitations of our small dataset. Although LLMs may offer advantages for larger-scale projects, Gumtree and SVM together provided a reliable and effective method for identifying and classifying bugs in our setting.

We conducted our analysis under the foundational assumption that well-engineered software code should follow best practices, such as maintaining unit tests for each functional component. Although this is not always strictly adhered to in practice, it remains a guiding principle for ensuring software quality and maintainability. Based on this, we assume that any modification to the program code should be accompanied by a corresponding modification to the test code. Specifically:

- If an existing function is modified, its associated test case should be updated to reflect the new logic or behaviour.
- If a new function is added, a corresponding new test case should be introduced to ensure it is tested.
- Conversely, if a function is deleted, its corresponding test case should also be removed to avoid leaving behind obsolete tests.

This principle formed the basis for our investigation into buggy and pre-buggy commits. We systematically compared these commit pairs to identify patterns and inconsistencies in the evolution of source code and test code. Our analysis focused on determining whether the changes were:

- Code change involving modifications to both program and test files, in which case we further examined whether the changes were:
  - Correlated e.g., a function modification accompanied by an updated or new test case.
  - Uncorrelated e.g., test files being modified independently of the program logic, or program logic changing with no test updates.
- Code with addition and deletion patterns is investigated to determine whether changes in one domain were mirrored appropriately in the other:
  - When new code was added, we checked whether new test cases were also added.

  - When code was deleted, we verified whether related test cases were similarly removed.

- Code changes confined solely to program files indicate a likely program bug or a missed test case and those limited only to test files indicate a test bug or a missing program change.

Given our limited dataset, we implemented 10-fold cross-validation to maximise the utility of available data and ensure reliable performance estimation. This technique involves partitioning the dataset into 10 distinct folds, with each fold serving once as the test set and the remaining 9 for training. This iterative process ensures that every data point is used for both training and validation, providing a comprehensive assessment of the model's generalisation capability. In each fold, we used 10% of the data for training and 90% for validation, allowing us to evaluate the model on previously unseen data and simulate real-world deployment scenarios. To assess the effectiveness of our approach, we applied a range of widely used machine learning algorithms, with the results detailed in the subsequent section. We also evaluated a small subset (5%) of the dataset using the publicly available OpenAI ChatGPT to compare its performance with that of conventional machine learning models. This subset was chosen because training LLMs effectively on such a limited dataset is challenging; our goal was to assess the LLM's performance when applied directly, without additional fine-tuning.

Our methodology is designed to emphasise generalisation. The features selected for classification are not specific to individual projects, but instead capture fundamental properties of code changes, enabling the model to adapt across diverse projects. Moreover, although program-related bugs are more prevalent in the dataset, we established a balanced class with dual distribution to avoid bias in training. This was achieved by down-sampling program bug instances, specifically by excluding samples from projects with disproportionately high bug counts. This strategy mitigates the risk of overfitting to dominant projects and ensures the inclusion of bugs from underrepresented projects, thereby promoting a more balanced, robust, and generalisable bug classification model.

7. Results

In this experiment, approximately 500 bugs were analysed, and the overall process consisted of three stages: data collection, AST extraction, and AST

comparison for bug classification. Before presenting the results, we highlight two primary aspects: (i) time efficiency, to verify that the approach remains feasible for integration into continuously running CI/CD pipelines, and (ii) a comparative analysis with LLM-based evaluation, to understand the relative value of the proposed ML-based approach and to assess the stability and consistency of its results when contrasted with an LLM.

To discuss the time efficiency and latency introduced by the bug classifier into the CI/CD pipeline, please refer to Table 1. In the first stage (data collection), the failed commit and its parent commit are located and downloaded, then checked out for analysis. This process took approximately 12 hours, with some variance attributed to project size, network interruptions, and timeouts (up to 30 minutes). The second stage (AST generation), which involved extracting AST features for all projects, required approximately 30 minutes. The final stage (bug classification), which performs AST comparison and classifies bugs using machine learning, took approximately 5 minutes. We also executed a subset of the dataset to classify using an LLM. The first two steps remained the same, but the final classification step was executed separately. This step took significantly longer to process in the LLM than in the machine learning–based approach.

Table 1: Execution Time for ML and LLM Based Bug Classification

Task Executed	**Time Overall (Mins)**	**Time Per Bug (Mins)**
GIT Download / Checkout	720	1.44
Extract AST	30	0.06
Bug Classification (ML)	5	0.01
Bug Classification (LLM: OpenAi GPT-4o)	*	0.25

Overall, the total processing time for each bug is less than 2 minutes in the ML-based approach, including the data download, which is the most time-consuming part. These results indicate that the proposed approach is computationally efficient and practical for CI/CD integration, without introducing substantial pipeline delays. Having established the computational feasibility of the proposed approach through time-cost evaluation, we now turn our attention to the ML and LLM based bug classification approach.

As outlined earlier, we employed ML techniques to classify bugs. In parallel, we tested a subset of LLMs to compare their performance with ML models and to determine whether this research makes novel contributions

beyond what LLMs can achieve. The objective was to evaluate consistency and feasibility in dynamic, continuous environments when operating with limited data.

Since the LLM is untrained, and we assume that training with limited data will not have a significant impact on the LLM, we tested the subset as-is on OpenAI (GPT-4o). Since the ML results are very consistent each time, we also wanted to test the LLM's performance multiple times to see if it's consistent. We selected a balanced subset of the dataset (5%) from the program and test, carefully chosen to avoid bias by excluding bugs from the same project, to simulate a realistic CI/CD pattern similar to what we used for ML. We chose the bugs and made a list ordered by type. Furthermore, we conducted 3 runs: (i) Top to Bottom, all program bugs first and test bugs later, (ii) Bottom to Top, all test bugs first and program bugs later in the reverse order, (iii) Mixed and Random, where we randomly mixed the bug types and in a random order. These were manually fed to the LLM, and the results were recorded refer Table 2. We observed that although the results had good predictive scores in each run, they weren't consistent across all three runs. The consistent accuracy is 50% lower than the Average score, and hence these results can't be trusted.

The LLM achieved results that were not too bad in individual runs, though its accuracy remained lower than that of the ML-based approach. However, its performance was not stable across the three experimental runs. The predictions that were both consistent and correct across all runs were approximately 50% lower than the average accuracy observed within each run. This highlights a notable degree of variability in the LLM's performance. This level of inconsistency suggests that the LLM results, in this untrained setting, are not reliable for practical bug classification tasks. The LLM achieved only 36% accuracy, compared to 73% for our ML-based experiments. Having established the performance advantage and reliability of the ML approach, we now proceed to address the research questions.

Table 2: LLM Based Bug Classification Across Multiple Runs: Accuracy and Consistency

Bug Type	**Run 1 (TB)**	**Run 2 (BT)**	**Run 3 (MR)**	**Consistent Accuracy**
Code	40.90%	27.27%	36.36%	13.36
Test	31.81%	45.45%	31.82%	22.72
Combined	72.72%	72.72%	68.18%	36.36

RQ1: *How effective are different machine learning models in classifying regression failures as program or test bugs across diverse projects?*

To answer this research question, we utilise SVM models [29], which are capable of providing robust classification even when the training data is limited and have proven performance with Boolean models. Among the techniques, the Histogram-based Gradient Boosting Classifier and MLPClassifier have generally performed better than the others. However, when evaluated separately for program bugs and test bugs, the Make Pipeline and SGD classifiers have performed the best.

Although a prior work provides a benchmark, the novelty of our generic bug-prediction methodology motivated us to establish an additional baseline tailored to this study. To achieve this, we constructed balanced samples of program and test bugs, ensuring that the Zero-R (0R) classifier yields a 50% accuracy baseline for each case. This balanced setup prevents bias toward the majority class in the training data and provides a fair, neutral reference point against which the performance of our models can be evaluated.

To evaluate the performance of our proposed methods, we compared them against this baseline. The results are shown in Table 3.

- Support Vector Classification (SVC): This model achieved a success rate of 64%. SVC is known for its robustness in finding the optimal hyperplane to separate classes, and its performance indicates a notable improvement over the baseline.

- Logistic Regression: This model is widely used, with a success rate of 60%. While it performs better than the baseline, its results are modest compared to some other methods, as its effectiveness can vary depending on the problem's complexity and the nature of the data.

- Extra Trees Classifier: This model performed the best among those tested, with an average success rate of 62%. The Extra Trees Classifier aggregates predictions from multiple decision trees, often improving performance over single decision trees and other classifiers.

- SGD: This model is widely used and has achieved a success rate of 57.5%. It works well with high-dimensional data, which may have contributed to its strong performance, given the dataset's many features.

- Pipeline: This model achieves an accuracy of 53% by ensuring consistent preprocessing steps, such as feature scaling with StandardScaler,

which is essential for SVMs sensitive to feature magnitudes. Properly scaled data might have allowed the SVM to perform optimally.

- HistGradientBoostingClassifier: This ensemble method uses histogram-based gradient boosting to speed up training by discretising continuous features into bins. While efficient, it achieved 67% accuracy, indicating potential struggles with complex patterns or the need for further tuning.

- MLPClassifier: A neural network model with multiple layers that learns complex patterns through nonlinear transformations. It achieved 67% accuracy, suggesting that it captures non-linearities in the data better than traditional methods, although further optimisation could improve its performance.

In contrast, several other machine learning models (Random Forest Regressor, Lasso Regression, SVR and Bayesian) were also applied but did not perform better than the Zero-R (0R) baseline. The fact that some models failed to surpass this benchmark suggests that they struggled to learn meaningful patterns from the data. This can occur for various reasons. Random Forest Regressor, despite being a powerful ensemble method, did not achieve performance levels higher than the baseline in our tests. This could be attributed to over-fitting or the specific characteristics of the dataset used. Lasso Regression struggled to exceed the baseline, potentially due to its inherent tendency to perform variable selection and regularisation, which may not have been well-suited to our specific dataset or feature set. SVR (Support Vector Regression) requires properly scaled features and may perform poorly if the kernel (e.g., linear) does not adequately capture the data's complexity, requiring careful parameter tuning. The Bayesian model could have performed poorly, as it tends to struggle with very small datasets and can yield inaccurate predictions if the initial assumptions about the data are incorrect.

In addition to the previously discussed machine learning models, we also incorporated the N-1 approach into our analysis. This method involves training the model on all instances except one, which is excluded from the training set in each iteration. The N-1 approach offers valuable insights into how well the model generalises to unseen data. The results from this approach show a similar trend to those from 10-fold cross-validation, reinforcing the model's consistency and reliability. By combining both methods, we gain a deeper understanding of the model's robustness in predicting bugs across various

scenarios, further complementing the baseline and other techniques we've used.

Table 3: Accuracy Scores for Program and Test Bug Predictions

ML Type	**Program%**	**Test%**	**Avg%**	**N-1%**
SVC	77	57	64	66
Log Regr	70	50	60	63
Xtree Clsf	62	62	62	66
SGD Classifier	48	67	57.5	62
Make pipeline	79	27	53	59
Hist Grd Boost (Ensemble)	75	59	67	59
MLPClassifier (Neural)	73	61	67	69

Our results indicate that different machine learning models exhibit strengths in classifying specific types of bugs. By strategically selecting the best-performing model for each category, such as Make Pipeline for program bugs and SGD Classifier for test bugs, we were able to harness their complementary capabilities refer Table 4. This approach ensures good classification accuracy across both bug types.

Table 4: Average of Best Performing ML Models

Best Performers	**Category%**	**Accuracy Score%**	**N-1 Score%**
Make Pipeline	Code bugs	79	59
SGD Classifier	Test bugs	67	62
Avg	*	73	60.25

RQ2 *Which structural features of code changes (derived from AST differencing) contribute most to distinguish program from test bugs?*

To address this research question, we have extracted structural features that reflect changes in the code, including additions and deletions of code elements (e.g., functions and classes). It includes changes like modifications to existing code, changes to control flow structures, patterns of variable usage, updates to test cases and a few others. These features were crucial for distinguishing between program bugs, which often arise from logical errors

in the code, and test bugs, which can result from outdated or ineffective test cases.

By generating ASTs for both buggy and corresponding pre-buggy versions of the code, we analysed the structural differences introduced in each commit. This AST-level comparison enabled us to precisely locate the syntactic and semantic modifications that could have contributed to the bug manifestation.

Our empirical results indicate that certain AST-derived features are strong predictors of bug type. Tables 5 and 6 present the top 10 most influential features identified and quantified using coefficients for the most efficient ML techniques for program and test bug classification, respectively. Interestingly, many top-ranked features were shared across both models and made a similar impact, suggesting that these features are robust and consistent indicators for bug classification. Notably, features involving modifications to test cases consistently ranked among the top contributors in both models, reinforcing their discriminative power in distinguishing between the program test bugs.

Table 5: Top features used to distinguish program bugs

Features	**Importance %**	**Description**
TFM-CM	16.28	Class modified in test file
TFM	14.02	Test file modified
CFM-CM-FA-TU	9.32	Relevant test cases updated for the function added in program file
CFM-CM-FM-TU	9.32	Relevant test cases updated for the function modified in program file
CFM-CM-FM	8.99	Function modified in program file
CFM	7.89	Program file modified
CFA	4.83	Program file modified
TFM-CM-FA	4.61	Function added in test file
CFM-CD	3.91	Class deleted in program file
CFM-CM	3.07	Class modified in program file

8. Threats to Validity

Size of dataset: Small datasets present inherent challenges in building a robust classification model and can also lead to overfitting, where the model might perform well on the training data but poorly on unseen data. Although we have utilised bugs from diverse projects to generalise the model's performance and have used only 10% of the data for training to predict 90% of

Table 6: Top features used to distinguish test bugs

Features	**Importance %**	**Description**
CFA	22.02	Program file modified
TFM-CM	11.73	Class modified in test file
TFM	10.26	Test file modified
CFM-CM-FA-TU	9.70	Relevant test cases updated for the function added in program file
CFM-CM-FM-TU	9.70	Relevant test cases updated for the function modified in program file
TFM-CM-FM	8.07	Function modified in test file
CFD	3.57	Program file deleted
TFA	3.23	Test file added
TFM-CM-FA	3.23	Function added in test file
CFM-CD	2.74	Class deleted in program file

the data, we recommend that future work should explore strategies for data augmentation or seek larger datasets to enhance the model's overall ability.

Scope of ML/LLM: This study focuses on conventional machine learning (ML) techniques for bug classification for two reasons. Firstly, most tools integrated within the CARF framework rely on ML-based prediction models. For instance, E-APR uses ML to recommend suitable repair tools, and feasibility-checking modules also employ ML to predict whether a tool can successfully fix a given bug. Since the ecosystem predominantly operates on ML methods, the proposed bug classifier naturally aligns with this architecture. Secondly, while Large Language Models (LLMs) such as CodeT5 [30], StarCoder [31, 32], and the GPT family have demonstrated strong capabilities in defect detection, they require substantial training data, high computational resources, and often suffer from reproducibility issues due to the use of proprietary APIs. Additionally, LLMs pose potential data leakage risks if test bugs are present in pretraining datasets, necessitating careful dataset curation. To investigate the behaviour and performance of LLMs, we conducted a small-scale comparison using a sample dataset with OpenAI GPT-4o against traditional ML models. The objective was to assess consistency and feasibility in continuous environments when working with limited data. Our results suggest that LLMs are less effective on smaller datasets compared to ML models. However, a thorough evaluation of LLM-based approaches would require a dedicated and more comprehensive study.

Data Quality and Verification: The dataset used in this study is

sourced from Bugswarm, and we have accepted the data at face value. We have not independently verified the accuracy of the labels provided in the dataset. This lack of validation could introduce inaccuracies if the labels are incorrect or inconsistently applied.

Flaky Tests: Our study does not consider the effects of flaky tests [33], and as a result, potential influences of flakiness on classification outcomes are outside the scope of this work. Although the BugSwarm already filters them out before including them in their dataset.

Framework: The proposed Continuous Automatic Repair Framework is a theoretical model that aggregates the capabilities of several existing tools and techniques and relies on the accuracy metrics reported in their respective publications. However, these reported results are often evaluated under controlled conditions and may not generalise across diverse contexts. Taking such reported accuracies at face value can lead to compounded errors when integrated, potentially making the end-to-end process less practical or reliable. Further empirical validation is required to assess the framework's real-world effectiveness and robustness.

9. Future works

Since the present classifier represents an initial step, a practical next move would be to begin implementing the CARF framework in a CI/CD setting to observe its behavior and identify potential challenges. Starting with this foundational work would provide valuable insights for gradually developing a more complete version of CARF and understanding its applicability in real-world APR scenarios.

A natural direction would also be to extend the bug classifier beyond the current binary distinction between program and test bugs. In real CI/CD environments, build failures can arise from a wide range of causes, including configuration errors, dependency issues, integration failures, environmental inconsistencies, and so many more. Enhancing the classifier to recognise these additional categories would make it more robust and operationally valuable.

The current design of the framework provides an architectural foundation for managing the APR life cycle and supporting the broader APR ecosystem. Its modular structure offers flexibility for future expansion, such as integration of additional processing channels and repair tools. For instance, a complexity assessment stage could be incorporated immediately after bug classification to distinguish between simple fixes and issues requiring more

nuanced repair strategies. Such a module could route simpler cases to tools such as PraPR[34], while directing more complex repairs to systems such as InferFix[35]. Such specialised extensions highlight the flexibility and customisability of CARF.

While this work developed a static bug classifier, an important direction for future investigation is the construction of a dynamic runtime instant bug classifier that can adapt to real-time pipeline feedback and evolving project contexts. A comparative study of static and dynamic classifiers may yield deeper insights into their respective trade-offs in accuracy, adaptability, and computational efficiency.

This work also needs to be generalised across additional programming languages. Such as extending the study to C++ (compiler-based) and Python (interpreter-based) systems would help determine whether the technique transfers effectively across language paradigms or whether language-specific adaptations and heuristics are required to maintain classification accuracy and operational efficiency.

10. Conclusion

In this research, we presented an initial step toward integrating APR tools into production environments such as CI/CD pipelines and identified the absence of an instant bug classifier as a critical missing component for realising this broader vision. Experiments on a diverse dataset of over 500 bugs from 67 projects demonstrate the effectiveness of our SVM models, achieving up to 79% accuracy for program bugs, 67% for test bugs, and an overall average of 73%, using only 10% of the training data outperforming baseline approaches and demonstrating strong generalisation. Our results are notable in two ways: (i) the method operates directly on readily available Git repository code without manual preprocessing, enabling a continuous and automated workflow; and (ii) it achieves higher success rates on a more diverse dataset than the closest comparable study, despite using far less training data (10% vs. 90%). These findings also suggest that as the modules access larger and richer datasets, performance will likely improve further, leading to increasingly refined and reliable results.

This study employed traditional machine learning (ML) techniques to develop the static bug classifier and contrasted its performance with an LLM-based baseline on the same dataset. It has been observed that ML techniques have proven more consistent, reliable, and accurate than LLMs. The reason

the proposed bug classifier relied on ML-based techniques was not only due to their robustness with limited data and low runtime overhead, but also for the fact that integrated tools and techniques in the CARF workflow employ similar ML-driven mechanisms, which made it easy to build the first prototype. But with the rapid advancements in AI and LLM technologies, the broader CARF ecosystem will eventually require a systematic transition from ML-based analyses to LLM-enhanced evaluation frameworks.

The availability of larger datasets can provide a richer experimental space for training and for investigating whether LLMs can be leveraged to construct bug classifiers with improved contextual understanding. Exploring hybrid approaches that integrate LLM embeddings into traditional ML could also offer a promising direction, as such hybridisation may capture deeper program semantics and enable more intelligent repair recommendations.

By extending the work outlined in the future directions, it would be possible to gradually develop a fully functional CARF framework. Such a framework could likely support continuous program repair in CI/CD pipelines, improve the reliability of automated software maintenance, and provide a foundation for further empirical evaluation in real-world settings.

11. Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used ChatGPT in order to improve the readability and language of the manuscript. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the published article.

References

[1] H. Eladawy, C. Le Goues, Y. Brun, Automated program repair, what is it good for? not absolutely nothing!, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024. doi:10.1145/3597503.3639095.
URL https://doi.org/10.1145/3597503.3639095

[2] V. Kabadi, D. Kong, S. Xie, L. Bao, G. A. Azriadi Prana, T.-D. B. Le, X.-B. D. Le, D. Lo, The future can't help fix the past: Assessing

program repair in the wild, in: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2023, pp. 50–61. doi:10.1109/ICSME58846.2023.00017.

[3] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, T. F. Bissyandé, A critical review on the evaluation of automated program repair systems, Journal of Systems and Software 171 (2021) 110817. doi:https://doi.org/10.1016/j.jss.2020.110817.
URL https://www.sciencedirect.com/science/article/pii/S0164121220302156

[4] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, D. Marinov, Reassert: a tool for repairing broken unit tests, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 1010–1012. doi:10.1145/1985793.1985978.
URL https://doi.org/10.1145/1985793.1985978

[5] D. Hao, T. Lan, H. Zhang, C. Guo, L. Zhang, Is this a bug or an obsolete test?, in: G. Castagna (Ed.), ECOOP 2013 – Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 602–628.

[6] K. Herzig, S. Just, A. Zeller, It's not a bug, it's a feature: How misclassification impacts bug prediction, in: 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 392–401. doi:10.1109/ICSE.2013.6606585.

[7] A. Aleti, M. Martinez, E-apr: Mapping the effectiveness of automated program repair techniques, Machine Learning 26 (5) (sep 2021). doi:10.1007/s10664-021-09989-x.
URL https://doi.org/10.1007/s10664-021-09989-x

[8] X.-B. D. Le, T.-D. B. Le, D. Lo, Should fixing these failures be delegated to automated program repair?, in: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), 2015, pp. 427–437. doi:10.1109/ISSRE.2015.7381836.

[9] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M.-Y. Wong, Orthogonal defect classification-a concept for in-process

measurements, IEEE Transactions on Software Engineering 18 (11) (1992) 943–956. doi:10.1109/32.177364.

[10] M. Monperrus, The living review on automated program repair, Tech. Rep. hal-01956501, HAL/archives-ouvertes.fr (2018).

[11] L. Huang, V. Ng, I. Persing, M. Chen, Z. Li, R. Geng, J. Tian, Autoodc: Automated generation of orthogonal defect classifications, Automated Software Engineering 22 (2011) 3–46.
URL https://api.semanticscholar.org/CorpusID:10258013

[12] F. Thung, D. Lo, L. Jiang, Automatic defect categorization, in: 2012 19th Working Conference on Reverse Engineering, 2012, pp. 205–214. doi:10.1109/WCRE.2012.30.

[13] X. Xia, D. Lo, X. Wang, B. Zhou, Automatic defect categorization based on fault triggering conditions, in: 2014 19th International Conference on Engineering of Complex Computer Systems, 2014, pp. 39–48. doi:10.1109/ICECCS.2014.14.

[14] F. Thung, X.-B. D. Le, D. Lo, Active semi-supervised defect categorization, in: 2015 IEEE 23rd International Conference on Program Comprehension, 2015, pp. 60–70. doi:10.1109/ICPC.2015.15.

[15] J. Xuan, H. Jiang, Z. Ren, J. Yan, Z. Luo, Automatic bug triage using semi-supervised text classification, ArXiv abs/1704.04769 (2017).
URL https://api.semanticscholar.org/CorpusID:3084096

[16] Neelofar, M. Y. Javed, H. Mohsin, An automated approach for software bug classification, in: 2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems, 2012, pp. 414–419. doi:10.1109/CISIS.2012.132.

[17] J. Hernández-González, D. Rodríguez, I. Inza, R. Harrison, J. A. Lozano, Learning to classify software defects from crowds: A novel approach, Appl. Soft Comput. 62 (2018) 579–591.
URL https://api.semanticscholar.org/CorpusID:35894602

[18] N. K. Nagwani, S. Verma, K. K. Mehta, Generating taxonomic terms for software bug classification by utilizing topic models based on latent dirichlet allocation, in: 2013 Eleventh International Conference on ICT

and Knowledge Engineering, 2013, pp. 1–5. doi:10.1109/ICTKE.2013.6756268.

[19] M. Leszak, D. E. Perry, D. Stoll, Classification and evaluation of defects in a project retrospective, J. Syst. Softw. 61 (3) (2002) 173–187. doi:10.1016/S0164-1212(01)00146-7.
URL https://doi.org/10.1016/S0164-1212(01)00146-7

[20] B. Freimut, C. Denger, M. Ketterer, An industrial case study of implementing and validating defect classification for process improvement and quality management, in: 11th IEEE International Software Metrics Symposium (METRICS'05), 2005, pp. 10 pp.–19. doi:10.1109/METRICS.2005.10.

[21] J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, X. Wu, Towards effective bug triage with software data reduction techniques, IEEE Transactions on Knowledge and Data Engineering 27 (1) (2015) 264–280. doi:10.1109/TKDE.2014.2324590.

[22] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, Y.-G. Guéhéneuc, Is it a bug or an enhancement?: a text-based approach to classify change requests, in: Conference of the Centre for Advanced Studies on Collaborative Research, 2008.
URL https://api.semanticscholar.org/CorpusID:214266

[23] M. Shahin, M. Ali Babar, L. Zhu, Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices, IEEE Access 5 (2017) 3909–3943. doi:10.1109/ACCESS.2017.2685629.

[24] M. Maes-Bermejo, A. Serebrenik, M. Gallego, F. Gortázar, G. Robles, J. M. González Barahona, Hunting bugs: Towards an automated approach to identifying which change caused a bug through regression testing, Empirical Softw. Engg. 29 (3) (May 2024). doi:10.1007/s10664-024-10479-z.
URL https://doi.org/10.1007/s10664-024-10479-z

[25] Z. Ni, B. Li, X. Sun, T. Chen, B. Tang, X. Shi, Analyzing bug fix for automatic bug cause classification, Journal of Systems and Software 163 (2020) 110538. doi:https://doi.org/10.1016/j.jss.2020.110538.

URL https://www.sciencedirect.com/science/article/pii/S0164121220300200

[26] Y. Zhao, H. Leung, Y. Yang, Y. Zhou, B. Xu, Towards an understanding of change types in bug fixing code, Inf. Softw. Technol. 86 (C) (2017) 37–53. doi:10.1016/j.infsof.2017.02.003.
URL https://doi.org/10.1016/j.infsof.2017.02.003

[27] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, C. Rubio-González, Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 339–349. doi:10.1109/ICSE.2019.00048.

[28] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: Proceedings of the International Conference on Automated Software Engineering, 2014, pp. 313–324. doi:10.1145/2642937.2642982.
URL https://hal.archives-ouvertes.fr/hal-01054552/file/main.pdf

[29] C. Cortes, V. Vapnik, Support-vector networks, Machine Learning 20 (3) (1995) 273–297. doi:10.1023/A:1022627411411.
URL https://doi.org/10.1023/A:1022627411411

[30] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, S. C. H. Hoi, Codet5+: Open code large language models for code understanding and generation, Machine Learning (2023). arXiv:2305.07922.
URL https://arxiv.org/abs/2305.07922

[31] R. Li, L. B. allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. LI, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, J. Lamy-Poirier, J. Monteiro, N. Gontier, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. T. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, U. Bhattacharyya, W. Yu, S. Luccioni, P. Villegas, F. Zhdanov, T. Lee, N. Timor, J. Ding, C. S. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M.

Ferrandis, S. Hughes, T. Wolf, A. Guha, L. V. Werra, H. de Vries, Starcoder: may the source be with you!, Transactions on Machine Learning ResearchReproducibility Certification (2023).
URL https://openreview.net/forum?id=KoFOg41haE

[32] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, H. de Vries, Starcoder 2 and the stack v2: The next generation (2024). arXiv:2402.19173.
URL https://arxiv.org/abs/2402.19173

[33] T. A. D. Henderson, B. Dorward, E. Nickell, C. Johnston, A. Kondareddy, Flake aware culprit finding, in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), 2023, pp. 362–373. doi:10.1109/ICST57152.2023.00041.

[34] A. Ghanbari, S. Benton, L. Zhang, Practical program repair via bytecode mutation, in: D. Zhang, A. Møller (Eds.), Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019, ACM, 2019, pp. 19–30. doi:10.1145/3293882.3330559.
URL https://doi.org/10.1145/3293882.3330559

[35] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, A. Svyatkovskiy, Inferfix: End-to-end program repair with llms, in: S. Chandra, K. Blincoe, P. Tonella (Eds.), Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023, ACM, 2023, pp. 1646–1656. doi:10.1145/3611643.3613892.
URL https://doi.org/10.1145/3611643.3613892