International Journal of Science and Research Archive

eISSN: 2582-8185

Cross Ref DOI: 10.30574/ijsra

Journal homepage: https://ijsra.net/

(Review Article)

AI driven defect prediction and automated refactoring framework for large scale software systems

Abdullahi Mohamud Hassan *, Ibrahim Rashid Abdullahi and Abdirizak Hussein Mohamed

Somali National university.

International Journal of Science and Research Archive, 2025, 17(02), 991-1004

Publication history: Received on 10 October 2025; revised on 15 November 2025; accepted on 18 November 2025

Article DOI: https://doi.org/10.30574/ijsra.2025.17.2.3098

Abstract

The importance of intelligent solutions that enhance software stability, maintainability, quality, and dependability has grown in recent years due to the increasing complexity of software systems. Conventional approaches to fault prediction and code rearrangement encounter scalability problems when confronted with large-scale, dynamic circumstances. The most recent advancements in automated refactoring frameworks and AI-driven fault prediction are examined in this review paper. In order to detect, prevent, and fix software problems before they happen, these frameworks use ML, DL, and NLP. Techniques to source code analysis, feature extraction, and quality improvement that rely on AI models are examined critically, along with emerging trends, methodology, and tools. Code structure optimization and automated refactoring decisions with minimal human interaction may be possible with the help of graph neural networks, RL, and predictive analytics, according to research. In addition, it explores the challenges of scalability, data quality, model interpretability, and integration with CI/CD workflows. Finally, the report concludes with research recommendations for future studies that could provide explainable, adaptive, and domain-independent frameworks to make software maintenance an autonomous, self-improving process.

Keywords: Automated Refactoring; Defect Prediction; Ai-Driven; Software Quality Assurance; Code Smell Detection; Large-Scale Systems

1. Introduction

This era of digital transformation has seen the rise of large-scale software systems, which have greatly increased the complexity of software creation and maintenance. Software with a large number of dependencies and millions of lines of code must be robust, extensible, and flexible. Enterprise platforms, systems native to the cloud, and cyber-physical systems all fall under this category. Continuous integration environments are utilized for the development of these apps. Due to their slowness, lack of precision, and lack of understanding of context, old software quality assurance approaches are ill-suited to the modern, fast-paced software engineering settings. Two of these approaches are defect prediction and manual rework.

System failures or costly maintenance costs are additional consequences of software system problems, along with decreasing functionality. Heuristic rules, statistical models, and human code inspections were the prior generation of defect prediction systems; nonetheless, they relied heavily on developer knowledge and databases of previous errors (Kathiresan, 2023). The methods used to build SQA were essential, but they are no match for the massive and ever-changing codebases found in modern systems. With the advent of AI came data-driven software restructuring, failure prediction, automation, and flexibility signaling a significant paradigm shift.

* Corresponding author: Abdullahi Mohamud Hassan

In order to anticipate possible vulnerabilities prior to deployment, artificial intelligence (AI)-driven defect prediction uses ML and DL approaches to detect components prone to faults, extract latent patterns from source code, and more. Several methods, including as RF, SVMs, CNNs, and GNNs, may faithfully portray software relationships and intricate code structures. Advances in natural language processing and transformer-based designs have brought models closer to becoming as intelligent as humans when it comes to understanding the nuances of source code's syntax and semantics. Automated refactoring frameworks powered by AI are also becoming more sophisticated and going beyond rule-based approaches. By evaluating dependency networks, identifying code smells, and examining earlier reworking trends, these systems may automate refactoring in a way that improves modularity, performance, and maintainability. Automated refactoring and defect prediction are two tools that intelligent, self-adjusting software maintenance pipelines can utilize to continuously boost quality.

However, there is still work that has to be done. Uneven data, unintelligible models, and unexplainable AI predictions all contribute to low adoption rates. Other challenges include integrating with DevOps and CI/CD environments. Because there aren't any established norms for evaluation, standardized datasets, or transparent decision-making frameworks, it is far more difficult to guarantee the repeatability and dependability of AI-based solutions in real industrial settings.

This review paper aims to assess the present status of AI-driven defect prediction and automated refactoring frameworks by examining recent theoretical and practical developments in the area. It highlights the possibility of new ideas, such reinforcement learning optimization, self-supervised models, and hybrid learning architectures, to enable software systems to evolve independently. In this preview, theoretical framework for solving the problems of large-scale software development in the age of AI-driven engineering: autonomous software quality assurance ecosystems. This is accomplished by drawing attention to gaps in our understanding and synthesising the findings of prior research.

2. Background and motivation

2.1. Traditional Defect Detection Methods

Code reviews performed by humans, rule-based testing, etc., are two popular approaches of finding bugs. The rate of current software engineering is increasing at a rate that these methodologies just cannot match. According to Maxim and Kessentini (2016), the traditional approaches take a lot of time and effort, and there's a high probability of human mistake because of how inefficient the process is for finding and fixing defects. Manual code reviews are great for finding syntax and logic mistakes, but they aren't consistent and can't handle very big projects because they rely on the reviewer's availability and skill (Rosenberg, 2003). The same holds true for rule-based testing; this approach is less effective in dynamic software settings because to its reliance on heuristics and static rules, which can make it blind to complicated context-specific problems. As software complexity rises and distributed systems become more common, it is more important than ever to enhance the present process with better ways to boost software dependability, since the challenges of defect identification are becoming worse. The use of cloud hosting, third-party integrations, and microservices designs has made traditional defect detection approaches inadequate for discovering these kinds of vulnerabilities and performance issues in modern software applications (Madhumita 2022). Traditional SQA procedures can produce scalability problems since, as is common with widely used systems, they require considerable testing, which is simply not feasible within the severely limited development period. Due to the fast growth of DevOps and agile approaches, software teams are now anticipated to release new versions often, sometimes even numerous times. In such cases, normal test and legally prescribed procedures will inevitably fail since they raise the percentage of defects that make it into production undiscovered. Nama et al. (2021) found that this causes software to fail and then requires costly patches. Humans' natural cognitive biases and limitations make manual fault detection a flawed process that can miss some major errors while highlighting others. There are always new frameworks, languages, and programming paradigms appearing in the software industry, and traditional metrics just can't handle it. Therefore, retraining is becoming more important for QA teams to maintain their efficiency (Hashi, 2025). Software companies are embracing AI-driven models that automate, employ deep learning, and machine learning to enhance software quality assurance, reduce testing times, and improve accuracy in defect identification in response to these limitations, as opposed to manually predicting defects.

2.2. Definition of Refactoring

The scholarly literature supports the general understanding that refactoring is making changes to code while preserving behavior (T. Mens, 2004). However, in reality, this definition appears to deviate significantly from the de facto standard. While Fowler does provide a list of 72 different kinds of structural modifications that can be made to object-oriented programs, it is not guaranteed that these changes would preserve the behavior. Since these refactorings can alter the behavior of a program, Fowler advises developers to write test code beforehand. In their analysis of refactoring logs. Johnson (2011) defined refactoring as follows: refactoring enhances behavior in certain areas but does not always maintain behavior in all areas.

Although there have been a number of studies that have helped us grasp the benefits of refactoring in a more theoretical sense, there have been very few empirical studies that have quantified those benefits. The initial connection between software modularity and option theories was made by Sullivan et al. (1998). A module also gives the possibility to substitute it with a superior one without symmetric commitments. Because it opens up great possibilities for module enhancement, Baldwin and Clark (1999) stated that system modularization can produce enormous value for an industry. Quick and dirty implementations leave technical debt, which leads to greater maintenance costs (J. Carriere 2010), a comparison that Ward Cunningham made between debt and lack of refactoring. Despite the fact that these efforts deepened our conceptual understanding of the effects of refactoring, they failed to measure those advantages.

The majority of Eclipse's structural changes—roughly 70%—have been refactorings, say Xing and Stroulia. and the IDEs that are available now aren't powerful enough to handle refactorings of this complexity. Using modularity criteria that were described by MacCormack et al., researchers were able to track the development of Linux and Mozilla. According to their findings, Mozilla's architecture is now much more modular than it was before the overhaul. But their research didn't pinpoint which modules underwent restructuring; instead, it only tracked changes to the design structure as measured by modularity measures, unlike our Windows study.

By comparing the software's coupling measurements before and after restructuring, Kataoka et al. (2002) presented an evaluation approach for refactoring. Kolb et al. (2006) found that software is more reusable and maintainable after restructuring in a case study that looked at the creation and release of existing software. Refactoring improves reusability and quality key performance indicators (KPIs), according to a case study by Moser et al. (2006) conducted in an industrial setting using agile methodology. The average time to resolve tickets was found to be reduced after the redesign of the system, according to Carriere et al. Ratzinger et al. discovered an inverse link between refactoring-related attributes and problems in their defect prediction models based on software evolution properties. In other words, a decrease in errors is expected when the number of refactoring adjustments grows relative to the prior time period. Reorganization leads to improvements in quality and production, according to these research.

Several approaches have been taken to solve the issue of automatically inferring refactorings from two versions of the program. In their 2006 study, Dig and Johnson found that by comparing code elements based on their names and structural similarities, these approaches may detect refactorings like relocating and renaming. The authors Prete et al. automate the process of finding complicated refactorings that occurred between two versions of the program through utilizing a logic query technique and encapsulating Fowler's refactoring types in prototype logic rules. You might find a more comprehensive list of all the currently available refactoring reconstruction methods elsewhere. Kim et al. (2011) explore the connection between API-level refactorings and mistake eradication by examining the outcomes of refactoring reconstruction. Instead of relying on refactoring reconstruction approaches, we use modifications extracted from branches that were generated to apply and maintain refactorings to discover refactorings in our Windows 7 study. Due to a designated team's manual verification of all refactoring branches and consensus on their respective roles, we are confident in the reliability of our refactoring identification technique.

2.3. Role of AI in Software Quality Assurance

There are various ways in which Quality Assurance (QA) and Software Testing can profit from and take advantage of AI, which is a major player in these fields. This section will focus on the function of AI in software testing and QA. In this article, we delve into the many facets of AI and its possible uses, advantages, and future prospects in quality assurance.

The overarching purpose of QA methods is to guarantee that the delivered goods and services are of a high enough standard. All aspects of it are designed to increase overall quality through thorough testing, monitoring, and continual improvement. In the past, quality assurance relied on labor-intensive and error-prone manual testing. Prior to the advent of AI, quality assurance lacked three key features: automation, accuracy, and scalability.

The most current development in artificial intelligence is the ability to predict and differentiate software defects. The most common AI-enabled defect prediction systems use analytics, deep learning methods, and automated machine learning models to improve defect detection, categorization, and repair procedures (Deming et al., 2021). To anticipate likely early failures or faults stemming from these histories, these defect prediction methods combine execution logs with metrics of code complexity, canopy patterns, and failure histories throughout the SDLC. Skilled and careful developers will check that everything is in order before deploying (Nama et al., 2021). AI-enabled systems increase fault prediction accuracy and perfect efficiency with less real human involvement, leading to a significant improvement in software quality (Pareek, 2021). Machine learning algorithms like as neural networks, SVMs, DT, RF, and others can deduce patterns in software defects that are challenging to discover using more conventional approaches. The same holds true for the deep learning method; specifically, RNNs and CNNs are used to discover hidden anomalies and intricate software design (Sohel and Begum, 2021).

Adding AI to a continuous integration and continuous delivery pipeline allows for a more proactive handling of vulnerabilities, leading to a more engaging development approach (Tyagi, 2021). Automating regression testing, enhancing test coverage in real-time, and prioritising test cases based on risk assessments are just a few of the numerous possible benefits of using AI for defect prediction. To maximize productivity, the community can use AI-powered predictive analytic models to zero in on the most impactful areas of software testing and development (Haghsheno, 2021).

New opportunities for cost reduction, faster software delivery, and improved reliability have emerged as a result of using AI for software quality assurance prediction. Data quality, model interpretability, and integration complexity are three major issues that come along with AI. The anticipated benefits of using AI into software testing and quality assurance will not materialize unless these are resolved. Improvements to AI models and explainable AI integration into underwear software development should be the subject of future studies.

3. AI-based defect prediction models

The foundation of modern SQA is no longer rule-based procedures or human metrics, but rather smart, data-driven models that can understand the semantic and structural nuances of source code. The scalability and predictive accuracy of defect detection systems have been greatly enhanced by recent breakthroughs in graph-based modeling, large-scale pre-trained code representations, and deep learning (DL). Akimova (2021), Giray (2022/2023), Pan et al. (2021), Cui et al. (2022), and others have recently detailed various methods for AI-based defect prediction. These methods utilize modern results from DL, hybrid modeling, transfer learning, and ML.

3.1. Machine Learning Approaches

For a long time, ML has been a vital tool in SDP. ML applies statistical inference and pattern recognition to software metrics such cyclomatic complexity, coupling, LOC, and code churn. The first ML-based models used a variety of techniques to identify problem modules and avoid them, including LR, SVMs, DT, Random Forests (RFs), and others. Despite the poor outcomes, these methods relied heavily on feature engineering, which hindered their ability to capture intricate connections inside code structures (Akimova, 2021).

Gradient Boosting (e.g., XGBoost) and Bagging are examples of ensemble methods that use a combination of many weak learners. These methods demonstrated improved generalization as the amount of available data rose. When machine learning methods aren't flexible enough to accommodate new projects and languages, a problem known as CPDP emerges. In WPDP, ML models achieve great accuracy because of distributional discrepancies between training and target projects. However, as pointed out by Javed et al. (2024), their performance suffers greatly in CPDP scenarios. An increasing number of initiatives are implementing domain adaptation and transfer learning strategies to normalize feature distributions (Springer, 2024). Despite these advancements, ML-based models are not without their limitations. We are developing more powerful deep learning architectures to automate the learning of latent code representations. Modeling code semantics and doing away with manually constructed measurements are both possible with these designs.

3.2. Deep Learning Techniques

Automated feature extraction and semantic representation learning from sources like raw code, commit histories, or ASTs have been game-changers for defect detection in DL algorithms. Akimova (2021) and Giray (2023) state that sequence-based, tree-based, and graph-based DL model designs have evolved throughout time.

3.2.1. Sequence-Based Models (CNN, RNN, LSTM)

To sequence models, code is only data expressed as text or tokens. Convolutional neural networks (CNNs) excel at local syntactic pattern recognition. Tasks necessitating the establishment of sequential or chronological links among code tokens, commits, or version histories are most effectively addressed by RNNs and LSTM networks. When it comes to JIT defect prediction, CNN-LSTM hybrids simulate the contextual evolution of code updates better than traditional ML classifiers, rendering to investigate by Zhao et al. (2023).

3.2.2. Graph-Based Models (GNNs)

Intriguingly, GNNs provide a new way to depict structural links and interactions. In their presentation of a GNN-based failure prediction framework, Cui et al. (2022) used graph nodes and edges to construct links between classes, modules, and functions. This method helps the model comprehend control flow, data dependence, and architectural interconnection, in contrast to conventional ML and DL models. Frameworks like Hierarchical Attention Graph (HAG-SDP) and Graph Sample and Aggregate were developed by researchers in 2024 and 2025 to improve the recording of code interactions at different levels.

3.2.3. Transformer-Based Models (CodeBERT, GraphCodeBERT, CodeT5)

A paradigm shift occurred with the introduction of transformer-based pre-trained models. Feng et al. (2020) presented CodeBERT, a model trained on programming and NLP concurrently, to better grasp code semantics and developer intent in a fully contextual manner. After undergoing fine-tuning and transfer learning, GraphCodeBERT and CodeBERT outperform typical DL models on cross-version and cross-project prediction tasks (Pan et al., 2021).

By utilizing self-attention techniques, these models based on transformers may be able to understand documentation signals, contextual semantics, and long-range code connections. According to Giray (2023), transformers are perfect for software analytics at the enterprise level due to their ability to accurately represent large-scale, heterogeneous repositories. On the other hand, hybrid and domain-adaptive methods are propelling the research forward. The reason behind this is that these models require extensive processing power and big tagged datasets.

3.3. Hybrid and Ensemble Methods

The interpretability of conventional ML is unparalleled, but the representational capacity of hybrid and ensemble techniques is on par with that of DL and transformers. Such solutions excel when working with complex industrial datasets, where different repositories may have very different types of flaws and where these problems can vary over time.

In a hybrid framework, you can use ML metrics (such process features or developer activity metrics) in conjunction with DL code embeddings (like those from CodeBERT or GNNs). This connection allows models to learn about both static and dynamic aspects of code quality. Compared to using only DL models, Zhao et al. (2023) demonstrated that combining CNN-derived features with hand-engineered metrics resulted in approximately 10% higher overall F1-scores.

To strengthen the system and make it less susceptible to overfitting, ensemble methods like stacking, boosting, and voting use several classifiers. To enhance the forecast accuracy in large open-source projects, Pan et al. (2021) utilized the stacked ensemble method. The ensemble comprised of CodeBERT embeddings and RF classifiers. Adversarial domain adaptation and meta-learning are two concepts that let models change their behavior across multiple projects dynamically. Employing Transfer-LSTM and DANN, Javed et al. (2024) proposed, will enhance CPDP's cross-domain generalization and decrease feature distribution inconsistencies. Similar to these hybrid systems, future frameworks for interpretable, scalable, and adaptable defect prediction will appear.

4. Automated refactoring techniques

4.1. Overview

In its original form, "refactoring" was proposed by Fowler (2019) and describes a methodical approach to improving the code's underlying structure and behavior. Its declared aims include lowering technological debt and raising program readability, extensibility, and maintainability. The ever-increasing scale and complexity of contemporary software systems render manual restructuring an inefficient, error-prone, and inconsistently demanding technique. Automation refactoring is becoming a popular topic among software engineers due to the fact that it allows for systematic and repeatable code changes (Baumgartner et al., 2024).

Automatic refactoring uses a mix of rule-based algorithms, heuristic search strategies, and AI-driven learning models to identify and fix "code smells" including god classes, extended methods, duplicated code, and data clumps. These systems ensure that refactoring operations maximize structural metrics and quality while keeping behavioral equivalence. Frameworks led by data-driven intelligence can autonomously maximize long-term maintainability by handling code semantics, inferred refactoring possibilities, and transformation priorities (Polu, 2025).

4.2. Code Smell Detection and Refactoring

There is a strong connection between code smell and future software problems and maintenance headaches. Automated refactoring greatly increases program lifecycles by discovering and correcting issues early on. Popular rule-based odor detection systems like PMD, SonarQube, and Checkstyle rely on static patterns and previously defined criteria for odor recognition. Techniques do not understand project-specific scenarios or semantic relationships, even while they work for basic syntactic problems (Uğur-Tuncer, 2020). Some recent AI-based methods get beyond these restrictions by combining supervised and unsupervised learning. Given labelled code samples, machine learning classifiers such as SVM, DT, and RF can detect complex smells like Large Class and Feature Envy. According to Baumgartner et al. (2024), context-aware smell detection can be enhanced by training deep learning models like GNNs and CNNs to understand the hierarchical and relational linkages found in code graphs and Abstract Syntax Trees (ASTs). With the help of these models, codebases may dynamically adjust by picking up on subtleties that were previously invisible. Clustering and autoencoders are examples of unsupervised algorithms that can be used to find structural anomalies in the absence of explicit labeling. Due of the dearth of labeled datasets, these methods work well for massive legacy projects. With the help of Explainable AI (XAI), which clarifies model decisions, developers can have faith in automated refactoring recommendations and transparency.

4.3. Evolutionary and Heuristic Algorithms

Alternatively, SBSE supports manual rewriting. This optimization-focused refactoring approach relies on objective functions as indices for coupling, coherence, complexity, and maintainability to assess possible solutions. Ouni et al. (2020) states that certain EAs, such as GAs, ACOs, and PSO, strive for refactoring sequences that are nearly perfect.

Both GAs and PSOs can find the best refactoring approaches by utilizing operators such as mutation and crossover, or by simulating social behavior. If you want to rearrange your courses, the way to do it well is with an ACO strategy, which relies on pheromone trails. Finding an optimal balance among performance, readability, and maintainability has never been easier than with these multi-objective optimization techniques.

One further intriguing method is the use of hybrid models that incorporate both heuristic optimization and machine learning for odor detection. By guiding search engines to solution spaces where ML models are more likely to succeed, these structures reduce processing costs and speed up convergence. Automated refactoring is more accurate and scalable when deep learning and evolutionary computation are employed combined, according to research by Polu (2025) and Baumgartner et al. (2024). In numerous domains of software, we have observed comparable advancements.

4.4. AI-Enabled Code Transformation

Recent developments in artificial intelligence (AI) have allowed automatic refactoring systems to understand and modify source code meaningfully. Learned from large code repositories, transformer-based models like CodeBERT, GraphCodeBERT, and CodeT5 define the syntactical and semantic relationships between code tokens. Models like these allow for the automated generation of refactoring code that follows best practices and adheres to design principles. New LLMs, including as OpenAI's GPT and Meta's Code Llama, do a good job of code reworking tasks like function extraction, code modularization, and variable renaming. . According to Thomas (2025), these models use few-shot or zero-shot learning to determine the goal of refactoring based on context clues like as comments, documentation, and usage patterns. They have also made use of RL techniques to progressively improve the refactoring quality. Here, an agent applies RL refactoring to a framework, monitors the impact on quality measures, and modifies its strategy according to incentive input.

Another interesting field is graphical code representation, which shows the source code of the code as a network of dependent variables, procedures, and variables. In dependency management and modularization, GNNs perform better than text-only models because they can inspect graphs for structural anomalies and propose suitable refactoring processes. As an example, these models make it possible for distributed or microservice-based systems to automate fine-grained decision-making.

4.5. Integration in CI/CD Pipelines

Automated refactoring is at its most effective when paired with solutions that enable continuous integration and deployment. Kathhiresan (2023) states that automated QA methods in DevOps environments include refactoring engines driven by AI. After analyzing incoming contributions and forecasting potential code degradation, these engines execute refactoring actions in real-time. The link between Jenkins and GitLab CI/CD ensures that quality requirements are achieved at every iteration by speeding code upload and refactoring activities.

In addition, AI models can be trained to adjust to project requirements that change over time by using continuous learning techniques. By providing the model with appropriate refactoring ideas, engineers can improve bad proposals and commend successful ones. This method integrates expert supervision with automation to improve model dependability and developer confidence. Anomaly detection, incremental learning, and knowledge distillation are some of the adaptive architecture technologies that will soon be integrated with autonomous refactoring to enable continuous integration and delivery. Continuous optimization with little to no human intervention is possible thanks to these systems' ability to self-adapt to varied codebases.

5. AI-driven defect prediction and refactoring frameworks

The goal of AI-driven frameworks like automatic refactoring and defect prediction is to find and fix code quality issues without errors. Toolschain compatibility, CI/CD, code models, and static analyzers are essential to the operation of such systems. We will discuss integration approaches and end-to-end pipeline patterns after introducing the key research and practice platforms.

5.1. Integration Strategies

Integrating automated refactoring methods and defect prediction models into AI-driven software quality pipelines is essential for ensuring reliability, transparency, and developer control. Two well-known integration models that have been employed in practice are the sequential predict-then-refactor strategy and the closely connected closed-loop approach.
A popular component of sequential integration methodology is the predict-then-refactor strategy. At the commit, file, or module level, the methodology's defect prediction component examines source code to generate risk ratings or localized defect warnings. Based on these forecasts, automatic refactoring recommenders suggest changes including renaming, method extraction, modularization, and code smell removal, among others. Developers have the option to select automated low-risk refactorings or human review based on their confidence level. By including clear audit trails for all recommendations, this design promotes accountability and openness. One example of an empirical application is the data-clump refactoring study that MDPI carried out in 2024. In order to identify problematic areas and trigger specific refactoring generators, this study suggests an AI-driven refactoring pipeline that utilizes GNNs and ML (Baumgartner, Tufano, Penta, and Kessentini, 2024).

There is an emphasis on continuous input and learning in closed-loop integration systems. Unit tests, static analysis, and symbolic equivalence checks are used to automatically analyze the refactoring procedures. A pipeline that can adapt to varied codebases and team dynamics is created by retraining or changing the defect prediction model employing the validation findings, demonstrating the relationship between the two components. Refactoring agents can be motivated to increase maintainability metrics without causing test failures by utilizing multi-agent architectures and reinforcement learning (RL), according to Karabiyik, Sharma, and Kesav (2025). By combining large language models (LLMs) with continuous integration and continuous delivery workflows, the RefactorGPT framework allows codebases to evolve autonomously while maintaining their semantic integrity (PeerJ, 2022).

There are three main considerations in the current solutions when thinking about integration from a practical perspective.

When dealing with production settings, it is important to prioritize confidence criteria and ways to keep people informed. So that only changes for which there is a higher and lower degree of risk are automatically implemented, conservative thresholds are established. When human validation is required, situations that are uncertain are marked as such. The use of SHAP-based feature attributions or transformer attention maps, which are examples of explainable AI (XAI) techniques, enhances developer trust and makes automated judgments more interpretable, according to Uğur-Tuncer (2020).

Another important consideration is the need for behavioral equivalence guarantees to maintain program semantics. To ensure that modifications do not impact functional behavior, regression, mutation, or symbolic equivalence testing is usually required in most frameworks (Baumeister et al., 2024) according to the literature. According to new research published in MDPI and ScienceDirect (Kathiresan, 2023), merging test development and refactoring validation can make large-scale CI/CD pipelines more reliable and reduce regression risk. Similarly, audit logging and traceability are crucial to rollback control and governance. Updated static analysis tools like as SonarQube and SonarCloud now include PR annotations, issue reporting, and repair suggestions. This establishes a direct link between the prediction of errors, code modification, and validation results (Rehman, 2025). Both compliance checks and retraining prediction algorithms can benefit from these trace logs. Systems that integrate predictive intelligence with continuous validation are becoming more popular in AI-powered refactoring and defect prediction frameworks. In these systems, you'll find improvements to auditability, explainability, and autonomy. This is critically important since the creation of self-repairing software systems is coming soon.

5.2. End-To-End AI Pipelines

The foundation of future sophisticated software quality assurance systems will be intelligent software that can automatically restructure code and anticipate software errors. It is feasible to apply permanent automation across large-scale software systems because to the intelligent transformation and continuing learning that these pipelines undergo after consuming raw code. Strict adherence to DevOps and CI/CD principles aims to preserve semantic integrity, scalability, and adaptability. Automated validation and refactoring with the help of artificial intelligence are two of the many proposed capabilities for modular frameworks that encourage codebase self-improvement (Baumeister et al., 2024; Karabiyik, 2025; Pan, Luo, and Zhang, 2021).

5.2.1. Data Collection and Preprocessing

An exhaustive data collection and processing strategy is required to construct an AI-powered refactoring and failure prediction system. A few examples of common data sources are commit histories, issue trackers, test outcomes, versioned repositories, and continuous integration logs. Features can't be extracted without code representations like CFGs, AST, and PDGs. Because models represent code, they can understand its syntactic and semantic parts (KATHhiresan, 2023).
Instead of testing models on future commits, training them on prior ones can help with defect prediction research and prevent knowledge leaking (Giray, 2023) instead. For accurate performance estimation, this method is necessary. Labeling techniques sometimes employ the utilization of commit-issue linkage to classify bug-fix contributions as defective modules. Code parsing, data standardization, and tokenization are some of the characteristics offered by databases like NASA MDP and PROMISE, according to Akimova (2021). To manage large-scale, dynamic codebases, machine learning models need to go through numerous steps of preprocessing.

5.2.2. Representation and Feature Extraction

During the representation learning stage, the code is converted into vectors or graph embeddings that machines can understand. Code churn, cyclomatic complexity, and lines of code were previously utilized in approaches (Thomas, 2025). In contrast, GNNs, CodeBERT (Feng et al., 2020), and GraphCodeBERT are examples of deep semantic encoders that have recently been developed that integrate syntax, lexicon, and structure information. Refactoring idea generation and defect localization graph representations are necessary for a successful AST to GNN conversion (Cui et al., 2022). All benchmarks were outperformed by hybrid encoders, which used a mix of GNN features for structural relations and transformer embeddings for token semantics. Pan et al. (2021) and Zhao et al. (2023) found that these models improve defect detection and downstream refactoring accuracy by providing context-aware and semantically rich embeddings.

5.2.3. Defect Prediction and Localization

An integral part of the pipeline, defect localization and prediction ascertains the probability of problems in certain modules or lines of code. There is a coexistence of older and newer model types in the field, including Random Forests, XGBoost, CNNs, RNNs, and transformer-based models. Akimova asserts that these models are capable of identifying data patterns that go beyond individual lines of code (2021).

Ensemble learning, which combines GNNs and transformers, can make your system more versatile and resistant to failure. The transfer learning capabilities of CodeBERT, which has been trained for cross-project defect prediction, are superior than those of traditional metric-based models (Pan et al., 2021). More and more, models also provide uncertainty estimates or confidence scores to determine if automatic refactoring is safe to execute. automatic CI/CD systems use these probabilistic outputs to make decisions in a reliable and automatic way (Baumgartner et al., 2024).

5.3. Tools and Platforms

Integrating research-oriented models with industry-standard tools that are scalable, semantically full, and reliably automated is a feasible strategy for AI-driven problem prediction and automated refactoring. Validation and deployment cannot be accomplished without industrial technologies like SonarQube and CI/CD platforms. Intelligent transformation capabilities and deep semantic comprehension are provided by advanced AI models such as GraphCodeBERT, CodeBERT, and GPT-based systems. For end-to-end pipelines to be able to assess, repair, and enhance themselves over time, all three must be in place for large-scale software systems.

5.3.1. SonarQube / SonarCloud and Sonar AI CodeFix

Modern software engineers rely on SonarQube and SonarCloud as essential tools for static code analysis, rule enforcement, and continuous inspection. In the past, SonarQube used a rule engine based on heuristics to detect vulnerabilities, defects, and code smells. To counter this, Sonar AI CodeFix has recently launched AI-assisted patch suggestions, enabling the platform to propose or implement fixes for specific types of bugs automatically (Rehman, 2025).

By integrating with CI/CD pipelines, SonarQube can prohibit merges when code quality is not up to par. That way, before they're integrated, any refactorings that AI modules automatically create are checked for maintainability, stability, and security. The REST APIs of SonarQube can do more than only validate; they can also activate the prediction-refactor-validation loop in AI-driven frameworks by sending out defect notifications (Baumgartner, Tufano, Penta, and Kessentini, 2024). Its annotation, logging, and visual issue tracking features are crucial for automated software pipeline maintenance that requires transparency and traceability.

5.3.2. Pretrained Code Models (CodeBERT, GraphCodeBERT, CodeT5)

Automatic refactoring, feature engineering, and SDP have all been transformed by using pre-trained code models that are generated by transformers with contextual semantic embeddings. According to Feng et al. (2020), CodeBERT is among the most significant designs among these. Models can be trained to comprehend code syntax and meaning by merging data from programmatic and natural language corpora.

Optimization of CodeBERT for SDP tasks significantly improves its ability to predict performance across versions and projects, as shown in the study by Pan, Luo, and Zhang (2021). An improved version of GraphCodeBERT is the data flow graph (DFG), which displays the interconnections and structural relationships between variables. This helps pinpoint the exact location of the issue and generates automated solutions to resolve it.

An improved transformer design for generative tasks, CodeT 5 enables the production of translations, refactorings, and direct code summaries (2021). In AI-driven pipelines, these pretrained models are used for refactoring generators and defect prediction modules, respectively. They synthesise transformations semantically and quantify risk. Furthermore, hybrid models like GNN-transformer ensembles incorporate their embeddings into the model, enhancing its ability to detect code smell, performance bottleneck, and maintainability issues (Cui, Zhang, and Liu, 2022; Giray, 2023).

5.3.3. Large Language Models and GPT-Based Systems

Improvements in code reorganization, summarization, and reasoning have been achieved through the new advancement of LLMs, such as the GPT series from OpenAI and comparable code-specialized models. Automated code transformations that can be understood by people depend on LLMs' ability to understand context, documentation in natural language, and overall design goals better than humans.

Research cited by Cordeiro (2024) indicates that LLMs like Codex and ChatGPT can enhance code readability, reorganize duplicate logic, and identify anti-patterns. Autonomous validation of their suggestions through test execution and static analysis is necessary because to the risks of hallucinations or semantic drift (Karabiyik, Sharma, and Kesav, 2025). In its multi-agent orchestration pipeline, the RefactorGPT framework employs LLMs as "transformation agents" to illustrate this point. It is the responsibility of the validator modules to check that the behavior and performance are compliant before any changes are made.

Ideas can be provided with confidence scores and reasons when LLMs are integrated into CI/CD and IDEs. This allows the developer-in-the-loop to participate. Giray (2023) argues that this approach is crucial for enterprise-level software engineering since it boosts productivity without doing away with human oversight.

5.3.4. Search-Based and Optimization Tools

By utilizing metaheuristic optimization techniques like PSO, Genetic Algorithms (GA), and Simulated Annealing, the Search-Based Software Engineering (SBSE) approach investigates possible refactoring orders. Uğur-Tuncer (2020) states that in order to achieve objectives like maintainability, performance, and cohesion, the aforementioned methods often employ ML-based defect predictors.

Effective editing of large search regions is made possible with hybrid learning, which incorporates AI into SBSE frameworks. The source is attributed to Baumgartner et al., 2024. To cut down on processing expense, models trained on SonarQube data can score potential refactorings before the search phase. By consistently boosting software modularity and minimizing regression, empirical evidence demonstrates that SBSE-based refactoring can beat manual approaches. A lot of end-to-end pipelines use SBSE in conjunction with LLM-driven refactoring modules because of SBSE's strength as a method for long-term code evolution. These modules generate context-aware, multi-objective refactoring plans by combining heuristic search with semantic reasoning.

5.3.5. CI/CD Integration (Jenkins, GitHub Actions, GitLab CI)

AI-driven refactoring procedures are built upon CI/CD, such as Jenkins, GitLab CI, and GitHub Actions. Automation of refactoring and defect prediction are made possible by these technologies, which streamline procedures that are activated by new pull requests or commits.

It is possible to invoke AI models as microservices for inference and validation utilizing GitHub Actions and Jenkins workflows, according to Rehman (2025). You can rest assured that all updates are evaluated against pre-set quality standards before being issued when you integrate with SonarQube. Advanced configurations that record developers accept/reject reactions to automated proposals through feedback collecting hooks allow AI models to be continuously retrained, according to Thomas (2025). Automation architecture that enables real-time quality assurance incorporates analysis, prediction, and controlled, verifiable automatic refactoring when required.

5.3.6. Test and Validation Tools

Validation frameworks must be strong for AI-driven automation to be reliable. It is possible to automate refactorings of programs without affecting their functionality by employing techniques for symbolic execution, unit testing, property-based testing, etc. Incorporating static analyzers like SonarQube or CodeQL into validation modules often results in a hybrid verification layer, as mentioned by Baumgartner et al. (2024). Static and dynamic testing are both encompassed in this tier.

Research published lately by the MDPI (Kathiresan, 2023) supports their claims that automated test development may enhance behavioral coverage in refactored programs. Constant input from validation findings not only guarantees dependability, but also allows retraining pipelines to enhance adaptive models. To make sure that self-learning refactoring frameworks are safe without sacrificing automation, validation tools operate as both teachers and guardians.

6. Challenges and research gaps

The industrial acceptance and generalizability of AI-driven defect prediction and automated refactoring are still hindered by a number of methodological, organizational, and technical issues, despite the noticeable advances in these areas.

6.1. Data Quality, Availability, and Label Imbalance

The majority of defect datasets that are made accessible to the public (such as PROMISE, NASA MDP, and AEEEM) are insufficient in size, have seen better days, and favor open-source Java systems. According to Giray (2023) and Javed et al. (2024), repeatability and cross-domain transfer learning are impeded when real-world industrial data is inaccessible owing to privacy and intellectual property problems. Inflated metrics and poor generalizability in practice are consequences of severe class imbalance, which occurs when defective modules make up less than 10% of samples (Akimova, 2021).

6.1.1. Model Generalization and Transferability

Domain drift, which includes differences in code style, architecture, and development practices, makes it such that models trained on one project rarely function well on others (Javed et al., 2024).

6.1.2. AI Systems' Explainability and Trustworthiness

Though realistic, deep and transformer-based models are difficult for developers to comprehend and depend on because to their "black boxes" nature (Liu et al., 2024).

6.1.3. Assurances of Safety and Semantic Validation

Behavioral equivalence cannot be guaranteed by automated refactoring systems. However, without regression tests or unfilled coverage gaps, it is still possible for incorrect transformations to pass validation gates, even with CI/CD pipelines and test suites in place (Baumgartner et al., 2024).

6.1.4. Integration Challenges and Humans in the Loop

Present continuous integration and continuous delivery systems do not facilitate the integration of refactoring tools and prediction modules. Clearly stated confidence criteria and the generation of observable outcomes are necessary for achieving an optimal automation/human supervision ratio (Rehman, 2025). Without enterprise-grade scalability and end-to-end deployment mechanisms, many frameworks are still in their early phases.

6.1.5. Moral and Environmental Considerations

There is a higher probability of licencing violations and data breaches when AI models use public code repositories. There are ethical concerns about the hardware needs and environmental impact of training large-scale models (LLMs) due to the high computing expenses involved (Cordeiro, 2024).

Research has shown that AI-driven refactoring is quite competent technically, but it needs to mature in areas like explainability, practical robustness, and human alignment before it can be used widely in businesses.

FUTURE DIRECTIONS

The goal of AI-powered SQA research should not be accuracy but rather the production of trustworthy, adaptive, and human-aligned solutions. Recent studies have shown that in order for software to adapt to new situations, continuous learning frameworks that use reinforcement adaption to update models on the fly are essential. By providing developers with interpretable forecasts and confidence-aware refactoring advice, explainable and responsible AI will increase transparency and trust. Through the use of LLMs, multi-agent architectures will be able to automate prediction, validation, and refactoring tasks in large software ecosystems in a coordinated fashion. Symbolic reasoning can be used in conjunction with LLMs to ensure semantic correctness while maintaining flexibility. To generalize across various repositories and programming paradigms, domain-adaptive and cross-language transfer models are essential. Lastly, AI has the potential to be a reliable co-evolving partner in software maintenance if we put an emphasis on long-term viability and promote human-AI cooperation to build frameworks that are developer-centric, lightweight, and environmentally efficient.

7. Conclusion

Artificial intelligence-driven defect prediction and automated refactoring frameworks have brought about a sea change in the software quality assurance industry. Now there is proactive, intelligent, and self-adaptive software creation instead of reactive maintenance. Newer, more advanced alternatives to modern statistical models include transformers, deep neural networks, LLM-based systems that integrate automatic correction with semantic comprehension, and others. These advancements have made it easier to scale, pinpoint problems more precisely, and integrate with continuous delivery pipelines. However, the potential of AI-driven engineering for completely autonomous and dependable software is still largely untapped. The necessity for continuous innovation is fueled by persistent issues including explainability, model drift, semantic validation, and cross-domain generalization. In order to guarantee safety, interpretability, and accountability, the best way to proceed is to incorporate human-centered, hybrid frameworks with symbolic reasoning, explainable artificial intelligence, and predictive modeling features. Software quality pipelines should include sustainable and green AI principles to further decrease computational costs and environmental effect while maintaining honesty and equity. Finally, SQA frameworks driven by AI have great potential to turn large-scale software maintenance into ecosystems that continuously improve themselves. In this model, AI would play the role of an ethically grounded, collaborative, and explainable partner, constantly predicting defects, suggesting optimal refactorings, and improving software resilience in the long run.

Compliance with ethical standards

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] Agnihotri, M., and Chug, A. (2020). A systematic literature survey of software metrics, code smells and refactoring techniques. Journal of Information Processing Systems, 16(4).

[2] Akhtar, S. M., Nazir, M., Ali, A., Khan, A. S., Atif, M., and Naseer, M. (2022). A Systematic Literature Review on Software-refactoring Techniques, Challenges, and Practices. VFAST Transactions on Software Engineering, 10(4), 93-103.

[3] Akimova, E. (2021). A survey on software defect prediction using deep learning. MDPI Electronics, 10(9), 1–25. https://doi.org/10.3390/electronics10091002

[4] Ali, M., Mazhar, T., Al-Rasheed, A., Shahzad, T., Ghadi, Y. Y., and Khan, M. A. (2024). Enhancing software defect prediction: a framework with improved feature selection and ensemble machine learning. PeerJ Computer Science, 10, e1860.

[5] Almogahed, A., Omar, M., and Zakaria, N. H. (2021). Empirical studies on software refactoring techniques in the industrial setting. Turkish Journal of Computer and Mathematics Education, 12(3), 1705-1716.

[6] Baldwin, C. Y., and Clark, K. B. (1999). Design rules: The power of modularity (Vol. 1). Cambridge, MA: MIT Press.

[7] Balogun, A. O., Odejide, B. J., Bajeh, A. O., Alanamu, Z. O., Usman-Hamza, F. E., Adeleke, H. O., ... and Yusuff, S. R. (2022, July). Empirical analysis of data sampling-based ensemble methods in software defect prediction. In International Conference on Computational Science and Its Applications (pp. 363-379). Cham: Springer International Publishing.

[8] Baumgartner, Nils; Iyenghar, Padma; Schoemaker, Timo; Pulvermüller (2024). AI-Driven Refactoring: A Pipeline for Identifying and Correcting Data Clumps in Git Repositories. Electronics, 13, 1644, 111129. DOI:10.3390/electronics13091644

[9] Cairo, A. S., Carneiro, G. D. F., and Monteiro, M. P. (2018). The impact of code smells on software bugs: A systematic literature review. Information, 9(11), 273.

[10] Carriere, J., Kazman, R., and Ozkaya, I. (2010). A cost-benefit framework for making architectural decisions in a business context. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE ’10) (pp. 149–157). New York, NY: ACM. https://doi.org/10.1145/1806799.1806821

[11] Cordeiro, J., Noei, S., and Zou, Y. (2024). An empirical study on the code refactoring capability of large language models. arXiv preprint arXiv:2411.02320.

[12] Cui, M., Long, S., Jiang, Y., and Na, X. (2022). Research of software defect prediction model based on complex network and graph neural network. Entropy, 24(10), 1373.

[13] Dig, D., and Johnson, R. (2005). The role of refactorings in API evolution. In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM ’05) (pp. 389–398). Washington, DC: IEEE Computer Society. https://doi.org/10.1109/ICSM.2005.73

[14] Dig, D., and Johnson, R. (2006). Automated detection of refactorings in evolving components. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP ’06) (pp. 404–428). Springer. https://doi.org/10.1007/11785477_25

[15] Ding, Y. (2024, October). Semantic-aware Source Code Modeling. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (pp. 2494-2497).

[16] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... and Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

[17] Giray, G. (2023). On the use of deep learning in software defect prediction: A systematic literature review. Journal of Systems and Software, 199, 111603. https://doi.org/10.1016/j.jss.2023.111603

[18] Gopinath Kathiresan. (2023). Leveraging Ai-driven defect prediction models for enhancing software quality assurance. Global Journal of Engineering and Technology Advances. 14(01), 136-148 DOI: https://doi.org/10.30574/gjeta.2023.14.1.0189

[19] Hodovychenko, M. A. H. M. A., and Kurinko, D. D. K. D. D. (2025). Analysis of existing approaches to automated refactoring of object-oriented software systems. Вісник сучасних інформаційних технологій, 8(2), 179-196.

[20] Javed, K., Shengbing, R., Asim, M., and Wani, M. A. (2024). Cross-Project defect prediction based on domain adaptation and LSTM optimization. Algorithms, 17(5), 175.

[21] Johnson, R. (2011, July). Beyond behavior preservation. Microsoft Faculty Summit 2011, Invited Talk.

[22] Karabiyik, M. A., Sharma, D., and Kesav, P. (2025). RefactorGPT: A multi-agent framework for automated code refactoring using large language models. PeerJ Computer Science, 11(2), 112–129.

[23] Kataoka, Y., Imai, T., Andou, H., and Fukaya, T. (2002). A quantitative evaluation of maintainability enhancement by refactoring. In Proceedings of the International Conference on Software Maintenance (pp. 576–585). https://doi.org/10.1109/ICSM.2002.1167826

[24] Kathiresan, G. (2023). Leveraging AI-driven defect prediction models for enhancing software quality assurance. Global Journal of Engineering and Technology Advances, 14(1), 136–148.

[25] Kim, M., Cai, D., and Kim, S. (2011). An empirical investigation into the role of refactorings during software evolution. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11) (pp. 151–160). IEEE/ACM. https://doi.org/10.1145/1985793.1985815

[26] Kolb, R., Muthig, D., Patzke, T., and Yamauchi, K. (2006). Refactoring a legacy component for reuse in a software product line: A case study. Journal of Software Maintenance and Evolution: Research and Practice, 18(2), 109–132. https://doi.org/10.1002/smr.325

[27] Hashi, A. I. (2025). Machine learning–driven fintech solutions for credit scoring and financial inclusion in the gig economy. International Journal of Innovative Science and Research Technology, 10(8), 1805–1820. https://doi.org/10.38124/ijisrt/25aug1023

[28] Li, Z., Niu, J., and Jing, X. Y. (2024). Software defect prediction: future directions and challenges. Automated Software Engineering, 31(1), 19.

[29] Liu, B., Jiang, Y., Zhang, Y., Niu, N., Li, G., and Liu, H. (2024). An empirical study on the potential of llms in automated software refactoring. arXiv preprint arXiv:2411.04444.

[30] MacCormack, A., Rusnak, J., and Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Management Science, 52(7), 1015–1030. https://doi.org/10.1287/mnsc.1060.0552

[31] Madhumita, R., and Chandana, D. (n.d.). Advancing software testing: Integrating AI, machine learning, and emerging technologies. Environments, 7(9).

[32] Maxim, B. R., and Kessentini, M. (2016). An introduction to modern software quality assurance. In Software quality assurance (pp. 19-46). Morgan Kaufmann.

[33] Mens, T., and Tourwé, T. (2004). A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[34] Murphy-Hill, E., Parnin, C., and Black, A. P. (2009). How we refactor, and how we know it. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09) (pp. 287–297). Washington, DC: IEEE Computer Society. https://doi.org/10.1109/ICSE.2009.5070529

[35] Nama, P. (n.d.). Intelligent software testing: Harnessing machine learning to automate test case generation and defect prediction.

[36] Pan, C., Luo, X., and Zhang, Y. (2021). An empirical study on software defect prediction using CodeBERT. Applied Sciences, 11(23), 11241. https://doi.org/10.3390/app112311241

[37] Prasad, R. D., and Srivenkatesh, M. U. K. T. E. V. I. (2025). A hybrid model combining graph neural networks, reinforcement learning, and autoencoders for automated code refactoring and optimization. Journal of Theoretical and Applied Information Technology, 103(1).

[38] Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M. (2010). Template-based reconstruction of complex refactorings. In Proceedings of the 2010 IEEE International Conference on Software Maintenance (pp. 1–10). IEEE. https://doi.org/10.1109/ICSM.2010.5609656

[39] Ratzinger, J., Sigmund, T., and Gall, H. C. (2008). On the relation of refactorings and software defect prediction. In Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR '08) (pp. 35–38). New York, NY: ACM. https://doi.org/10.1145/1370750.1370757

[40] Rehman, A. (2025). AI Driven Code Review System: Leveraging Artificial Intelligence for Enhanced code Quality Assessment and Bug Detection.

[41] Rosenberg, L. H. (2003). Lessons learned in software quality assurance. In A. Aurum, R. Jeffery, C. Wohlin, and M. Handzic (Eds.), Managing software engineering knowledge (pp. 251–268). Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-45141-8_14

[42] Sullivan, K., Chalasani, P., and Sazawal, V. (1998). Software design as an investment activity: A real options perspective [Technical report].

[43] Thomas, A. (2025). Review of AI-Driven Approaches for Automated Defect Detection and Classification in Software Testing. International Journal of Recent Research, Volume 12; Issue: 6 E-ISSN: 2349-9788; P-ISSN: 2454-2237

[44] Uğur-Tuncer, G., Davutoglu, C., and Durakbasa, N. M. (2020, September). Intelligent test automation for improved software quality assurance. In The International Symposium for Production Research (pp. 614-624). Cham: Springer International Publishing.

[45] Wang, S., Liu, T., Nam, J., and Tan, L. (2018). Deep semantic feature learning for software defect prediction. IEEE Transactions on Software Engineering, 46(12), 1267-1293.

[46] Xing, Z., and Stroulia, E. (2005). UMLDiff: An algorithm for object-oriented design differencing. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05) (pp. 54–65). New York, NY: ACM. https://doi.org/10.1145/1101908.1101919

[47] Zhao, Y., Damevski, K., and Chen, H. (2023). A systematic survey of just-in-time software defect prediction. ACM Computing Surveys, 55(10), 1-35.