2026 Volume: 02 Issue: 01 www.ijrpa.com ISSN 2456-9995 Review Article

International Journal Research Publication Analysis

Page: 01-32

# THEORETICAL REVIEW ON DEEP LEARNING ALGORITHM FOR AUTOMATED CODE REVIEW AND BUG DETECTION

**[*1]Chika L. Onyagu, [2]Odii Maduabuchi [3]Ezeamasiobi Chibuzor E., [4]Chekwebe Nwankwo**

[1]Department of Cybersecurity, Delta State University, Delta State, Nigeria.

[2] Department of Computer Science, Nnamdi Azikiwe University, Awka, Nigeria.

[3] Department of Computer Science, African University of Science and Technology, Abuja.

[4] Department of Computer Science, Chukwuemeka Odumegwu University, Uli, Nigeria.

**Article Received: 08 December 2025**

**Article Revised: 28 December 2025**

**Published on: 16 January 2026**

***Corresponding Author: Chika L. Onyagu**

Department of Cybersecurity, Delta State University, Delta State, Nigeria.

DIO: https://doi-doi.org/101555/ijrpa.8639

## ABSTRACT

This theoretical review examines the application of deep learning algorithms in automated code review and bug detection, highlighting their potential to improve software quality and development efficiency. Traditional static and dynamic analysis tools often struggle with complex code patterns and require extensive manual tuning. Deep learning models, particularly recurrent neural networks, convolutional architectures, and transformer-based language models, provide data-driven approaches capable of learning semantic and syntactic relationships directly from source code. These models enable automated identification of bugs, code smells, security vulnerabilities, and stylistic inconsistencies with higher accuracy and adaptability. The review also discusses key challenges, including the need for large, high-quality labeled datasets, handling diverse programming languages, and ensuring model interpretability for developers, emphasized by many scholars in this domain. Despite these limitations, deep learning continues to advance automated code analysis, offering promising directions for intelligent development environments and continuous integration pipelines. Overall, the theoretical foundations suggest substantial benefits for future software engineering practices.

**KEYWORDS:** Theoretical Review, Deep learning, Algorithm, Automated code review, Bug detection.

## 1.1 INTRODUCTION

Software Quality Assurance (SQA) refers to a systematic set of activities designed to ensure that software products and development processes meet defined quality standards across the entire software development life cycle (SDLC). It encompasses planning, design, implementation, deployment, and maintenance, emphasising preventive practices aimed at reducing defects, improving reliability, and ensuring that software aligns with user requirements. Rather than relying solely on defect detection, SQA incorporates proactive, quality-oriented processes such as audits, metric-driven evaluations, standards compliance, and continuous monitoring based on frameworks like ISO 9001, CMMI, and IEEE guidelines to enhance consistency, predictability, and long-term process improvement. These practices also ensure systematic evaluation of core software quality attributes, including functionality, performance, reliability, maintainability, portability, and usability.

Within this broader framework, code review remains one of the most essential SQA practices. Manual code reviews enable early identification of defects, enforcement of coding standards, and assessment of design decisions. Despite their benefits, manual reviews are often time-consuming, inconsistent, and heavily dependent on the expertise and availability of reviewers. As modern software grows in scale and complexity, traditional review methods increasingly struggle to keep pace with the rapid volume of code changes, limiting their effectiveness in large, fast-evolving systems.

These limitations have led to the emergence of automated and intelligent techniques for software quality assurance. Traditional Static Analysis Tools (SATs), linters, and rule-based engines offer increased efficiency in detecting syntactic issues, stylistic violations, and common error patterns. However, such tools remain constrained by their reliance on predefined rules and shallow pattern matching, often failing to capture deeper semantic relationships within source code or identify subtle context-dependent bugs (Li et al., 2018). These weaknesses have motivated a growing shift toward data-driven and learning-based approaches capable of modelling complex program behaviour.

Recent advancements highlight the transformative potential of AI-driven and deep-learning-based automation frameworks in SQA. Kavuri (2025) emphasises that modern AI-enhanced systems integrate machine learning (ML), natural language processing (NLP), and deep learning (DL) to automate critical testing tasks such as test case generation, execution, defect prediction, and test script maintenance. Through reinforcement learning, these frameworks can continuously improve test coverage and adapt to evolving software behaviour. NLP-powered models further enable the conversion of human-readable requirements into

executable test scripts, significantly reducing manual effort and accelerating validation processes. Additionally, AI-based defect analytics offer predictive capabilities that allow teams to prioritise high-risk components, reducing test cycle duration and overall human workload (Kavuri, 2025).

Deep-learning-based vulnerability detection systems further illustrate the capabilities of intelligent SQA tools. For example, Li et al. (2018) introduced *VulDeePecker*, a system that uses code gadgets, semantically related code segments, to detect subtle vulnerabilities that traditional static analysis tools often miss. Their experiments demonstrated that deep learning can substantially reduce false negatives and identify real-world vulnerabilities that were silently patched but never reported, showcasing the promise of learning-based approaches for improving automated code review and bug detection.

Despite these advancements, challenges remain. AI-driven SQA systems depend heavily on high-quality datasets, face interpretability concerns, and often require careful integration into modern CI/CD pipelines (Kavuri, 2025). However, the growing body of research, including deep-learning-based vulnerability detection, automated code-review recommendation systems, and ML-based defect prediction frameworks, demonstrates a clear movement toward intelligent, scalable, and context-aware quality-assurance tools (Li et al., 2018).

In summary, SQA establishes the essential foundation for developing reliable and high-quality software systems. While manual code review continues to play a critical role, its inherent limitations underscore the need for more advanced, adaptable, and automated approaches. The integration of AI, ML, and deep learning into SQA represents a paradigm shift toward more efficient, intelligent, and robust code review and bug detection processes. These emerging technologies not only enhance software dependability but also accelerate release cycles and support the rapidly evolving demands of modern software development environments (Kavuri, 2025; Li et al., 2018).

Code review is a fundamental practice in software development aimed at improving code quality, ensuring compliance with coding standards, and reducing the likelihood of defects in software systems. It involves the systematic examination of source code by one or more developers other than the original author to identify issues such as logic errors, inefficiencies, security vulnerabilities, and poor design decisions. As a core component of Software Quality Assurance (SQA), code review serves as an early checkpoint in the development workflow, enabling developers to detect and address defects before they propagate into later stages of the software life cycle (Fregnan, Petrulio, Di Geronimo, et al., 2022).

Several techniques have traditionally been used in conducting code reviews, each with its own strengths and limitations. One of the oldest and most formal approaches is the Fagan Inspection, a structured review process involving multiple stages such as planning, overview, preparation, inspection meeting, rework, and follow-up. This technique is highly effective in identifying deep structural and logical defects but is often time-consuming and resource-intensive. A more flexible approach is peer review, where developers examine each other's code informally, either synchronously (e.g., face-to-face review sessions) or asynchronously (e.g., comments on version control platforms). Peer reviews encourage knowledge sharing and team collaboration but may vary in effectiveness depending on reviewer expertise and availability (Fregnan, Petrulio, & Bacchelli, 2022).

Modern development practices have introduced additional techniques such as pair programming, where two developers work together at the same workstation, one writes code while the other reviews in real-time. This method enhances code quality and fosters collaborative problem-solving, although it may increase development cost. Another widely used method is tool-assisted code review, supported by version control systems and collaborative platforms such as GitHub, GitLab, and Gerrit. These platforms enable asynchronous review, inline comments, automated checks, and integration with Continuous Integration/Continuous Deployment (CI/CD) pipelines, making the review process more efficient and scalable (Fregnan, Petrulio, Di Geronimo, et al., 2022).

Underlying these techniques are key principles that guide effective code reviewing. One essential principle is readability, which emphasises that code should be easy to understand, well-structured, and properly documented so reviewers can easily identify issues. Another key principle is maintainability, requiring code to be modular, consistent, and adaptable to future changes. Consistency with established coding standards and project guidelines ensures uniformity, reduces technical debt, and facilitates collaborative development. Additionally, good review practices emphasise constructive feedback, where reviewers provide clear, actionable, and respectful comments aimed at improving the code rather than criticising the developer (Fregnan, Petrulio, & Bacchelli, 2022).

Despite the value of traditional techniques, their manual nature poses challenges in large-scale or fast-paced software environments. Manual reviews can be slow, inconsistent, and susceptible to human error or oversight. Moreover, as systems grow in complexity, understanding intricate logic or detecting subtle bugs becomes increasingly difficult, even for experienced reviewers. Empirical studies indicate that review outcomes often involve evolvability concerns, with documentation and structural changes being most common, and

that the number of review changes is influenced by factors such as patch size and added lines of code rather than reviewer comments alone (Fregnan, Petrulio, & Bacchelli, 2022). These findings highlight the complexity and limitations of manual review processes and further motivate the need for automated and intelligent systems.

In modern software engineering, automated code review tools and deep learning-based systems are emerging as powerful solutions for addressing these limitations, offering faster, more consistent, and more accurate analysis of source code. Such systems can automatically classify review changes, providing information that is perceived by practitioners as valuable for improving the code review process (Fregnan, Petrulio, Di Geronimo, et al., 2022). These advancements complement traditional practices and represent an important evolution in how software teams ensure high-quality code.

Software bug detection refers to the process of identifying errors, defects, or anomalies within a software system that cause incorrect or unexpected behaviour. Effective bug detection is crucial for ensuring software reliability, security, and performance. Over the years, various approaches, ranging from manual methods to advanced automated techniques, have been developed to detect bugs early in the development cycle. These approaches can generally be categorised into manual, static, dynamic, and intelligent (machine learning–based) techniques (Akhtar et al., 2023; Shaon & Akter, 2025).

One of the earliest methods for bug detection is manual code review, where developers examine source code line-by-line to identify logical errors, inconsistencies, and violations of best practices. Although this method can uncover deep structural issues, it is time-consuming, susceptible to human bias, and limited in scalability. To address these limitations, automated techniques have gained prominence, with static and dynamic analysis being the two most widely adopted approaches.

Static bug detection involves examining source code without executing it. Tools such as SonarQube, PMD, FindBugs, and ESLint apply rule-based and pattern-matching techniques to identify potential issues such as null pointer dereferences, unused variables, unreachable code, code smells, and security vulnerabilities. Static analysis is particularly effective at detecting syntactic and structural issues early in development and can be integrated into Continuous Integration (CI) pipelines for automated quality checks. However, rule-based static analysis tools often suffer from limitations such as high false-positive rates and an inability to interpret complex program logic or understand deeper semantic relationships (Akhtar et al., 2023; Shaon & Akter, 2025).

In contrast, dynamic analysis detects bugs by executing the software and observing its runtime behaviour. This category includes techniques such as unit testing, integration testing, fuzz testing, symbolic execution, and runtime monitoring. Dynamic analysis is effective for identifying issues such as memory leaks, race conditions, buffer overflows, and API misuse. Tools like Valgrind, AFL (American Fuzzy Lop), and JUnit support various aspects of dynamic testing. While dynamic analysis can uncover runtime-specific defects that static tools miss, it requires executable code, comprehensive test coverage, and often significant computational resources (Akhtar et al., 2023).

Beyond traditional static and dynamic methods, advancements in artificial intelligence have given rise to machine learning (ML) and deep learning (DL)-based bug detection approaches. These approaches leverage historical code data, bug reports, and code change patterns to learn statistical relationships between code characteristics and the presence of defects. Supervised learning models, for instance, classify code segments as buggy or clean based on extracted features such as complexity metrics or code tokens. Meanwhile, deep learning models, including recurrent neural networks (RNNs), convolutional neural networks (CNNs), transformers, and graph neural networks (GNNs), can automatically learn semantic and structural representations of code without extensive manual feature engineering (Akhtar et al., 2023; Shaon & Akter, 2025). These intelligent models have demonstrated improvements in detection accuracy and reduced the need for manually defined rules.

Another emerging approach involves hybrid bug detection, which combines static analysis, dynamic testing, and machine learning to leverage the strengths of each method. For example, hybrid models can use static analysis outputs as features for training machine learning classifiers or utilise runtime traces to enhance deep learning performance. This integrated strategy enhances precision and recall, particularly for complex bugs that require both structural and behavioural understanding (Akhtar et al., 2023).

Despite progress in these techniques, bug detection remains a challenging task due to evolving software complexity, increasing codebases, and the need to detect subtle logic or security vulnerabilities. Traditional rule-based tools struggle with scalability and contextual understanding, while machine learning approaches require large, high-quality datasets and may face issues such as model interpretability, class imbalance, and limited vulnerability coverage. Furthermore, modern approaches, including Large Language Models (LLMs), provide significant potential for capturing both syntactic and semantic properties of code but introduce challenges such as hallucination and high computational cost (Shaon & Akter, 2025). These limitations motivate ongoing research into more advanced, intelligent, and

automated methods, including neuro-symbolic hybrid approaches, parameter-efficient fine-tuning, cross-language generalisation, continual learning, and explainable AI for vulnerability detection (Shaon & Akter, 2025).

Machine Learning (ML) has emerged as a transformative technology in software engineering, enabling automated analysis, prediction, optimisation, and decision-making across various phases of the software development lifecycle. ML techniques leverage data-driven patterns to understand software artefacts, detect anomalies, recommend improvements, and support tasks that traditionally require human expertise. As modern software systems become increasingly complex and data-intensive, ML provides a scalable and intelligent approach to addressing long-standing challenges related to code quality, bug detection, maintenance, and developer productivity (Khalid et al., 2023; Yadav et al., 2024).

In software engineering, ML applications can be broadly classified into tasks such as defect prediction, code classification, effort estimation, automated testing, code recommendation, and software documentation. These tasks rely on diverse data sources, including source code, commit histories, code metrics, bug reports, execution logs, and developer interactions. ML models learn from these artefacts to perform predictions or classifications that assist developers in making informed decisions. For instance, defect prediction models analyse historical bug data and software metrics to estimate the likelihood of defects in new code modules, enabling teams to allocate testing and review resources more effectively (Khalid et al., 2023).

ML encompasses multiple paradigms, each applicable to different software engineering problems. Supervised learning is commonly used when labelled datasets exist, such as code segments labelled as "buggy" or "clean." Models such as decision trees, support vector machines (SVM), random forests, and neural networks are often employed for defect prediction, code smell detection, and vulnerability classification. Unsupervised learning techniques, including clustering and anomaly detection, help discover hidden patterns in code repositories or identify unusual behaviour in system logs. Reinforcement learning is also being explored for applications such as automated program repair, compiler optimisation, and adaptive testing, where the model learns optimal actions through iterative interactions with the environment (Khalid et al., 2023; Yadav et al., 2024).

A critical component of ML in software engineering is feature engineering, which involves extracting meaningful representations of code to transform it into a format suitable for ML models. Traditional approaches rely on manually crafted features such as cyclomatic complexity, lines of code, coupling, cohesion, and naming conventions. While these features

are useful, they often fail to capture the deeper semantics and structural relationships in code. To overcome these limitations, representation learning and deep learning techniques have gained prominence, enabling models to automatically learn complex features from raw code data (Yadav et al., 2024).

ML has significantly enhanced automated code review by enabling systems to analyse code patterns, understand developer intent, and generate context-aware feedback. ML-based models can learn from historical review comments, commit messages, and review outcomes to assist reviewers by predicting potential issues, suggesting improvements, or prioritising code segments requiring attention. Similarly, in bug detection, ML algorithms can identify recurring defect patterns and generalise beyond predefined rules, offering more flexible and scalable solutions than traditional static analysis tools (Khalid et al., 2023).

Research demonstrates that optimised ML models can achieve very high accuracy in defect prediction. For example, SVM and optimised SVM models have achieved accuracies up to 99% and 99.8%, respectively, while other models, such as Naive Bayes, Random Forest, and ensemble approaches, also perform strongly (Khalid et al., 2023). In code smell detection, ML has also proven effective, with algorithms like SVM, J48, Naive Bayes, and Random Forest being widely applied to identify early warning signs of potential software quality issues. These approaches help developers detect structural or design problems during the coding phase, supporting higher software quality and maintainability (Yadav et al., 2024).

Despite its advantages, the integration of ML into software engineering faces challenges such as data scarcity, imbalanced datasets, noisy labels, language diversity, and model interpretability. Software evolution introduces concept drift, requiring models to adapt continuously to maintain performance. Nonetheless, ML continues to play a central role in modern software engineering, providing the foundation for more advanced methods such as deep learning and hybrid approaches that further enhance automated code review and bug detection systems (Khalid et al., 2023; Yadav et al., 2024).

Deep Learning (DL) is a subfield of Machine Learning that focuses on neural networks with multiple hierarchical layers capable of automatically learning complex patterns from data. Unlike traditional ML approaches that rely heavily on manual feature engineering, deep learning models extract high-level abstractions directly from raw input, making them particularly powerful for tasks involving unstructured data such as images, speech, natural language, and increasingly, source code. DL's ability to model semantic relationships and nonlinear dependencies has positioned it as a leading technique in modern artificial intelligence research and applications (Mienye & Swart, 2024).

At the core of deep learning lies the artificial neural network (ANN), which consists of interconnected layers of neurons organised into input, hidden, and output layers. Each neuron performs a weighted transformation of its inputs followed by a nonlinear activation function, enabling the network to approximate complex functions. As the number of hidden layers increases, the network gains the capacity to learn deeper and more abstract features. Training is achieved using backpropagation and gradient-based optimisation algorithms such as Stochastic Gradient Descent (SGD), Adam, or RMSprop (Mienye & Swart, 2024).

Several specialised deep learning architectures have been developed to handle different types of data and tasks. Convolutional Neural Networks (CNNs) are well-suited for grid-like data and have been widely applied in computer vision. Their ability to learn spatial features through convolutional operations has also proven useful in source code analysis, particularly when treating code as token sequences or structural graphs. Recurrent Neural Networks (RNNs), along with their variants Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), are designed for sequential data and can capture long-range dependencies. These models have been used in tasks such as code generation, bug prediction, and automatic comment generation, where understanding the sequential nature of code tokens is essential (Mienye & Swart, 2024).

A major breakthrough in deep learning came with the introduction of the Transformer architecture, built around the concept of self-attention. Transformers, including models such as BERT, GPT, and their code-specific adaptations (CodeBERT, GraphCodeBERT, CodeT5), excel at capturing contextual relationships by attending to all parts of an input sequence simultaneously. This parallelism enables them to process long sequences efficiently and learn richer semantic representations of code. Transformers have thus become a dominant architecture in state-of-the-art automated code review and bug detection systems (Mienye & Swart, 2024).

Graph Neural Networks (GNNs) have also emerged as a crucial tool in deep learning for code analysis, particularly for tasks that require a structural understanding, such as bug localisation. Source code naturally forms structures like Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and data dependency graphs. GNNs propagate information through graph nodes and edges, enabling the model to capture both syntactic structure and semantic interactions. This makes GNNs particularly effective for detecting logical bugs, security vulnerabilities, and structural anomalies. Recent work has demonstrated that graph-based bug classifiers can accurately identify buggy nodes in code graphs, effectively

localising a wide range of bug types, including undefined properties, functional errors, variable naming errors, and variable misuse (Yousofvand et al., 2026).

Training deep learning models typically requires large datasets, high computational power, and careful tuning of hyperparameters such as learning rate, batch size, number of layers, and regularisation techniques. Methods such as dropout, batch normalisation, and early stopping help prevent overfitting and ensure that models generalise well to unseen data. Transfer learning is increasingly used, allowing pre-trained models to be fine-tuned on domain-specific code datasets, significantly reducing training costs and improving performance for specialised tasks (Mienye & Swart, 2024).

Despite these advantages, deep learning models face several challenges. They often require substantial data to achieve high performance, and their internal mechanisms can be difficult to interpret, a concern in software engineering where explainability is critical. Poorly trained models may struggle with code from unfamiliar programming languages or unconventional coding styles. Nonetheless, the strengths of deep learning, particularly its ability to learn semantic, contextual, and structural patterns, make it a powerful foundation for automated code review, bug detection, and code understanding (Mienye & Swart, 2024).

Representation learning for source code refers to the process of transforming program elements, such as tokens, syntax structures, control flows, and dependency relationships, into numerical representations that machine learning and deep learning models can effectively process. Unlike natural language, source code is highly structured, governed by strict grammar rules, and contains deep semantic dependencies. As a result, effective representation learning is fundamental to enabling automated code review, bug detection, vulnerability analysis, code summarisation, and various other intelligent software engineering tasks.

Traditional machine learning techniques rely on manually engineered features derived from code metrics, token frequencies, or structural characteristics. While these handcrafted features offer limited insights, they fail to fully capture the rich semantics and hierarchical structure embedded in modern programming languages. Contemporary approaches overcome these limitations by learning distributed representations that encode both syntactic and semantic information in dense vector spaces.

A foundational line of work treats source code as token sequences, similar to natural language. Embedding techniques such as Word2Vec, GloVe, and FastText have been adapted to generate vector representations for tokens, identifiers, and keywords, capturing contextual relationships useful for classification or comment generation tasks. However, token-based

models often struggle to capture deeper structural dependencies, as the meaning of code extends beyond linear token order.

To better capture hierarchical structure, many approaches incorporate Abstract Syntax Trees (ASTs), which represent the syntactic organisation of code. Models such as TreeLSTM and other recursive neural architectures leverage the parent–child relationships in ASTs to extract structural information and better understand program logic. A notable advancement in AST-based learning is *code2vec*, which represents a code snippet by decomposing it into multiple paths in its AST and learning embeddings for these paths jointly. These path-based representations are aggregated to form a single fixed-length vector capable of predicting semantic properties, such as method names (Alon et al., 2018). The ability of code2vec to learn representations from millions of methods illustrates the effectiveness of structural decomposition in capturing semantic regularities across large codebases.

Beyond syntactic structure, more expressive representations incorporate graph-based semantics. Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs) model execution order and variable interactions, respectively, enabling deeper analysis of program behavior. Graph Neural Networks (GNNs), including GCNs, GATs, and MPNNs, have been widely adopted to learn from these graph structures, capturing both local and global semantic dependencies essential for tasks such as bug detection and vulnerability discovery. Recent work further expands this paradigm by integrating multiple forms of program graphs into unified models. For example, CogCol converts code graphs into unique sequences and applies supervised contrastive learning to strengthen structural understanding and improve generalisation across similar code patterns, addressing the limitations of purely syntactic AST-based approaches (Shi et al., 2024).

Pre-trained language models for code have also transformed representation learning. Trained on large-scale repositories, these models combine token-level and structure-level information to produce context-aware embeddings that reflect both syntactic and semantic relationships. Their effectiveness across tasks like automated code review, defect prediction, and code retrieval highlights the power of large-scale pre-training for capturing deep code semantics.

An emerging direction is multimodal representation learning, which integrates multiple views of code, including tokens, ASTs, CFGs, DFGs, and execution traces, into a unified embedding. These multimodal models leverage complementary structural and semantic information to achieve greater robustness and improved performance across diverse software engineering tasks.

Despite these advancements, several challenges persist. Differences in syntax and semantics across programming languages complicate cross-language generalisation. Detecting subtle or logic-dependent bugs often requires modelling complex program dependencies that may not be fully represented in generic embeddings. In addition, labelled datasets for bug detection and defect prediction remain limited, hindering the training of high-capacity models. Nonetheless, ongoing research in structural, semantic, and multimodal representation learning continues to push the boundaries of automated code understanding, providing a strong foundation for advanced intelligent systems in software engineering.

Deep learning models for automated code review aim to assist developers in identifying defects, improving code quality, and ensuring compliance with software engineering standards through intelligent, machine-driven analysis. Traditional automated review tools depend heavily on handcrafted rules and heuristics, which are effective at detecting syntactic issues but struggle to capture deeper semantics and contextual logic. Deep learning provides a powerful alternative by learning complex patterns from large codebases and enabling models to reason about structural dependencies, functional intent, and semantic relationships in source code. Recent studies highlight that graph-based and deep neural models significantly outperform conventional static analysis tools in vulnerability detection and semantic understanding (Abdul Kadar, 2022).

Various neural architectures have been explored for automated code review, each offering unique strengths. Sequence-based models such as RNNs, LSTMs, and GRUs treat code as token sequences and learn contextual dependencies across statements. These architectures have been applied to tasks such as predicting review comments, detecting code smells, and identifying stylistic inconsistencies. However, because they rely on sequential token representations, these models often struggle with long-range dependencies and the rich structural complexity inherent in modern programming languages (Yin et al., 2023).

Transformer-based models have become the dominant approach for code intelligence due to their ability to capture global context through self-attention mechanisms. Pretrained models such as CodeBERT, CodeT5, PLBART, GraphCodeBERT, and CodeGPT have demonstrated state-of-the-art performance across code review tasks by learning powerful semantic and contextual representations from massive open-source repositories. These models support fine-tuning for domain-specific code review scenarios, allowing high accuracy even with relatively small datasets. For example, models that fuse structural information, such as program dependency graphs, with sequence-based representations within transformer architectures have shown notable improvements in accuracy and robustness (Yin et al., 2023).

Graph-based deep learning models represent another important paradigm in automated code review. By expressing software as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Data Flow Graphs (DFGs), or Program Dependency Graphs (PDGs), Graph Neural Networks (GNNs) capture rich semantic and structural information that sequential models often overlook. Studies have shown that GNN-based representations significantly enhance vulnerability detection, providing notable gains in accuracy, context awareness, and reduction of false positives compared to traditional tools and earlier neural approaches (Abdul Kadar, 2022). Program dependency graph serialisation methods, such as PDG2Seq, further improve representational quality by converting complex semantic graphs into unique sequences while preserving structure and meaning, enabling models like CodeBERT-based architectures to more effectively detect and correct defects (Yin et al., 2023).

Hybrid and multimodal deep learning approaches combine multiple representational views, including token sequences, ASTs, CFGs, and PDGs, to achieve more robust automated code review. These models often integrate transformers with GNNs to capture both semantic context and structural dependencies, improving the detection of subtle issues such as variable misuse, logical inconsistencies, or resource mismanagement. The fusion of sequence and structural representations has proven especially valuable for tasks requiring nuanced reasoning, such as automated fix suggestion or context-aware comment generation.

Deep learning has also enabled systems to generate natural-language review comments derived from historical pull request discussions and developer feedback. These models can articulate issues and propose improvements in human-readable form, reducing cognitive load and enhancing the collaborative review process. Attention mechanisms further improve interpretability by highlighting influential regions of the code, addressing concerns about transparency and model explainability, both important in professional software engineering contexts.

Despite substantial progress, challenges remain. Deep learning models require large, high-quality datasets containing code and corresponding review annotations; yet, these datasets are difficult to curate due to privacy constraints, inconsistency in review styles, and the labour-intensive nature of labelling. Moreover, the opaque inner workings of deep neural models raise concerns about trust and explainability, especially when automated feedback influences production systems. Ensuring that automated code review systems provide reliable, actionable, and transparent insights remains an ongoing research priority.

Deep learning has significantly transformed the landscape of software bug detection by enabling automated systems to learn patterns of defective code directly from large datasets.

Unlike traditional rule-based or heuristic methods, deep learning approaches possess the ability to capture complex semantic relationships, structural dependencies, and contextual patterns within source code. This makes them particularly effective for detecting subtle bugs, logic errors, and security vulnerabilities that may not be easily identifiable through static analysis tools or manual code review. Recent work also shows that deep learning models benefit from considering not only code features but also inter-module dependencies, as treating software systems as interconnected graphs can yield improved defect prediction performance (Cui et al., 2022).

One of the earliest deep learning approaches for bug detection involves sequence-based models, where source code is treated as a sequence of tokens similar to natural language. Models such as Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), and Gated Recurrent Units (GRUs) are commonly used in this paradigm. These models learn long-range dependencies within code, allowing them to detect patterns associated with common bug types, such as incorrect API usage or logical inconsistencies. Although effective in modelling token-level context, sequence-based methods sometimes struggle with structural complexity, as raw token sequences cannot fully capture the hierarchical and graph-oriented nature of source code. This limitation has also been observed in modern issue-tracking datasets, where text-based bug reports require richer structural modelling to improve prediction accuracy (Siachos et al., 2025).

To overcome the limitations of sequential processing, convolutional neural networks (CNNs) have also been employed in bug detection tasks. CNNs, while traditionally used in image analysis, can be adapted to operate on encoded representations of code, such as token embeddings or serialised abstract syntax trees (ASTs). Their strength lies in detecting local patterns, enabling the identification of small but critical code fragments associated with defects. However, CNNs are less effective when deeper semantic understanding or global context is required, especially for complex bugs involving long-range dependencies or multi-module interactions.

A breakthrough in deep learning for bug detection came with the introduction of transformer-based architectures, which use self-attention mechanisms to model global dependencies within code sequences. Models such as CodeBERT, CodeT5, GraphCodeBERT, and DeepBugs have achieved state-of-the-art performance in numerous bug detection tasks. Transformers excel in capturing the contextual relationships between variables, function calls, and control flows, making them highly effective for identifying complex logical bugs and security vulnerabilities. Pre-training on massive code corpora allows these models to

generalise across programming languages and defect types. Fine-tuning on bug-specific datasets further improves predictive performance by addressing domain-specific characteristics.

Another influential direction in deep learning-based bug detection is the use of Graph Neural Networks (GNNs). Many software bugs arise from improper data flows, variable misuse, or broken control paths, patterns that are naturally represented as graph structures. GNNs operate on representations such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Data Flow Graphs (DFGs), and Program Dependency Graphs (PDGs), propagating information across nodes and edges to capture both syntactic and semantic properties of code. This graph-centric perspective aligns with emerging research that models entire software systems as complex networks, treating classes or modules as nodes and their dependencies as edges, enabling more accurate defect prediction through improved structural representations (Cui et al., 2022). Additionally, hybrid graph-text models leveraging Graph Attention Networks (GATs) have demonstrated strong performance in predicting bugs from textual issue descriptions by combining semantic embeddings with graph-based relationships (Siachos et al., 2025).

In addition to standalone architectures, hybrid deep learning models integrate multiple representations of code—tokens, ASTs, CFGs, embeddings, and execution traces—to provide a more robust understanding of program behaviour. Such models leverage the complementary strengths of different views to detect subtle bugs that might be missed by single-representation approaches. For instance, a hybrid model may combine a transformer to capture high-level semantics with a GNN to analyse data dependencies, yielding more accurate predictions of variable misuse or incorrect control flow.

Moreover, deep learning has enabled anomaly detection approaches, in which autoencoders or variational autoencoders (VAEs) learn latent representations of “normal” code behaviour and detect deviations that may signify defects. These methods are particularly valuable when labelled datasets are scarce, allowing unsupervised or semi-supervised learning to identify unusual patterns resembling potential bugs.

Despite the remarkable progress, deep learning approaches for bug detection face challenges such as data imbalance, limited availability of high-quality labelled datasets, and difficulties related to model interpretability. Software systems vary widely in language, architectural style, and programming practices, making generalisation across domains difficult. Furthermore, developers often require transparent explanations for identified bugs, yet deep models typically operate as black boxes, complicating real-world adoption.

Nevertheless, deep learning continues to push the boundaries of automated bug detection, offering scalable, accurate, and intelligent solutions that complement human expertise and traditional tools. With ongoing advancements in graph-based modelling, multimodal learning, and transformer architectures, deep learning is expected to play an increasingly central role in next-generation software quality assurance systems (Cui et al., 2022; Siachos et al., 2025).

Evaluation metrics play a critical role in assessing the performance, reliability, and effectiveness of automated code review and bug detection systems. These metrics provide quantitative measures that help researchers and practitioners determine how well a model identifies defects, classifies code segments, generates review comments, or supports decision-making during software development. Choosing appropriate metrics ensures fair comparisons between techniques and provides insights into their strengths and limitations (Albattah & Alzahrani, 2024).

For classification-based tasks, such as distinguishing buggy from non-buggy code, commonly used metrics include Accuracy, Precision, Recall, and F1-Score. Accuracy measures the proportion of correct predictions made by the model; however, it becomes less reliable when datasets are imbalanced, which is often the case in bug detection, where non-defective code typically outnumbers defective code. In such cases, a model could achieve high accuracy while failing to detect actual defects. To address this, Precision and Recall offer more nuanced evaluation. Precision measures the proportion of correctly identified buggy instances among all predicted buggy instances, which is essential when minimising false positives is a priority. Recall measures the proportion of actual buggy instances correctly detected, reducing false negatives and ensuring critical defects are not overlooked. The F1-Score, the harmonic mean of Precision and Recall, balances these concerns, providing a single metric that reflects overall predictive quality (Albattah & Alzahrani, 2024).

The Confusion Matrix is often employed to provide a comprehensive view of model performance by summarising true positives, true negatives, false positives, and false negatives. It enables deeper analysis of model behaviour and facilitates identification of specific error patterns. Similarly, the Receiver Operating Characteristic – Area Under the Curve (ROC-AUC) evaluates the trade-off between true positive and false positive rates across varying thresholds. ROC curves are particularly useful for defect prediction models, as they provide an overall assessment of classifier performance across all possible threshold values, helping to identify optimal operational points for practical use (Morasca & Lavazza, 2020).

When automated code review systems generate natural-language comments or suggestions, evaluation metrics shift toward language quality and semantic relevance. Metrics such as BLEU (Bilingual Evaluation Understudy), ROUGE (Recall-Oriented Understudy for Gisting Evaluation), and METEOR assess the similarity between generated and reference comments, analysing aspects such as n-gram overlap, word precision and recall, and semantic alignment. Although originally developed for machine translation and summarisation, these metrics are now standard for evaluating textual feedback in code review environments (Albattah & Alzahrani, 2024).

For ranking or prioritisation tasks, such as recommending which files or lines require urgent attention, metrics like Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) are employed. These assess a model's ability to correctly rank buggy or problematic code higher, which is crucial for helping developers focus their efforts efficiently. In industrial contexts, additional metrics related to software quality may be considered, including reductions in post-release defects, review latency, developer productivity, and maintainability indices. Though harder to quantify, these metrics reflect the real-world impact of automated review and bug detection tools (Albattah & Alzahrani, 2024).

Selecting appropriate metrics depends on the goals of the system being developed. For instance, a bug detection model aimed at minimising missed defects may prioritise Recall, whereas a static analyser integrated into a continuous integration pipeline may emphasise Precision to reduce unnecessary alerts. Similarly, natural-language comment generation models rely on linguistic metrics, while structural defect detection models depend on classification metrics such as Accuracy, F1-Score, or ROC-AUC (Morasca & Lavazza, 2020; Albattah & Alzahrani, 2024).

Despite significant progress in applying deep learning to automated code review and bug detection, several challenges and limitations continue to constrain the effectiveness, reliability, and adoption of these systems. These challenges stem from the complexity of software systems, the evolving nature of programming languages, and inherent limitations in machine learning methodologies. Understanding these issues is essential for guiding future research and improving system performance (Viswanadhapalli, 2024).

One major challenge is the inherent complexity and variability of source code. Unlike natural language, code is highly structured and governed by strict grammatical and semantic rules. Small changes in syntax can drastically alter program behaviour, making it difficult for deep learning models to capture exact semantics. While modern models effectively learn statistical patterns, they may fail to understand deeper logic, data flow, or interactions between

components, leading to incorrect predictions or shallow analyses. This limitation becomes even more pronounced in large-scale or highly modular projects where context extends across multiple files (Viswanadhapalli, 2024; Golovnev, Starovoytov, & Staroletov, 2025).

Another critical limitation is the scarcity of high-quality, well-labelled datasets. Although numerous code datasets exist, many suffer from label noise, incomplete bug descriptions, or inconsistent annotations. For bug detection tasks, the imbalance between buggy and non-buggy samples often causes models to exhibit poor recall, failing to detect rare but significant defects. Furthermore, datasets sourced from open-source repositories may not be representative of proprietary or domain-specific software, limiting the generalizability of trained models (Viswanadhapalli, 2024; Golovnev et al., 2025).

The dynamic and evolving nature of software development introduces additional challenges. Programming languages, frameworks, and libraries are continually updated, causing models trained on older data to become outdated. Emerging coding patterns, new vulnerabilities, and evolving best practices require ongoing retraining and dataset updates. Failure to adapt can result in outdated or irrelevant recommendations, reducing trust in automated systems (Viswanadhapalli, 2024).

Interpretability is another significant concern. Deep learning models, particularly large neural architectures, often operate as black boxes, providing little insight into why a specific piece of code is flagged as defective. Developers require clear, interpretable explanations to trust automated suggestions. Without transparency, these systems may be perceived as unreliable or overly cautious, which can limit adoption in professional environments (Viswanadhapalli, 2024; Golovnev et al., 2025).

False positives and false negatives present additional obstacles. Excessive false positives can overwhelm developers with unnecessary warnings, reducing productivity and discouraging tool usage. Conversely, false negatives may allow critical defects to remain undetected. Balancing precision and recall is especially challenging for complex bugs that require deep semantic reasoning or contextual understanding across multiple files (Viswanadhapalli, 2024).

Integrating automated code review tools into existing development workflows also presents difficulties. Many organisations involve multiple stakeholders, diverse tools, and varying coding standards. Automated systems must be highly adaptable to different project structures, codebases, and review cultures. Poor integration can disrupt workflows, create redundancy, or conflict with human reviewers' judgments (Golovnev et al., 2025).

Deep learning models also demand substantial computational resources for training and inference. Large models can be expensive to train and maintain, particularly in continuous integration pipelines where rapid feedback is essential. Resource constraints may prevent smaller teams or organisations from deploying advanced models, limiting their practicality (Viswanadhapalli, 2024).

Detecting semantic, logical, or context-dependent bugs, such as concurrency issues, race conditions, or security vulnerabilities that depend on runtime behaviour, remains particularly difficult. Static code alone may not reveal these defects, and current deep learning models struggle to infer dynamic behaviour without execution traces or symbolic analysis tools (Viswanadhapalli, 2024; Golovnev et al., 2025).

Ethical concerns also arise, including potential bias embedded in training data, which may cause models to favour certain coding styles, patterns, or developer practices. This can lead to non-uniform treatment of contributions and inadvertently introduce unfairness (Viswanadhapalli, 2024).

Finally, developer trust and acceptance continue to be major barriers. Developers may resist tools that generate unclear, incorrect, or overly aggressive suggestions. Building trust requires consistent performance, interpretability, and the ability to complement rather than replace human expertise (Viswanadhapalli, 2024; Golovnev et al., 2025).

In summary, while deep learning has significantly advanced automated code review and bug detection, challenges ranging from dataset limitations and interpretability issues to workflow integration and computational constraints continue to limit the effectiveness and scalability of these systems. Addressing these challenges is essential for developing robust, trustworthy, and practical solutions capable of supporting modern software engineering processes (Viswanadhapalli, 2024; Golovnev et al., 2025).

### 2.1 Review of Related Works

The increasing complexity of software systems has made manual code review and bug detection both time-consuming and error-prone, prompting significant research into automated approaches. Traditional static and rule-based analysis tools, such as SonarQube and FindBugs, have been widely used to detect code smells and potential bugs, but they often fail to capture deeper semantic or context-specific issues. Consequently, researchers have turned to deep learning techniques to improve automated code analysis. Recent studies have explored sequence-based models, such as LSTMs, to learn patterns from historical code changes and suggest corrections or detect anomalies. More recently, transformer-based

models, including CodeBERT and GraphCodeBERT, have leveraged large code corpora and structural representations to achieve state-of-the-art performance in tasks like code review, bug detection, and code summarisation. While these deep learning approaches show considerable promise, challenges remain in terms of dataset requirements, generalisation across diverse projects, and computational cost. This section reviews related work in deep learning-based automated code review and bug detection, with particular attention to models that capture semantic and structural aspects of source code.

**Siva et al (2023). Automatic software bug prediction using adaptive artificial jelly optimisation with long short-term memory.**

Siva et al. (2023) proposed a deep learning-based framework for software bug prediction, aiming to improve software quality and reliability by detecting defects at early stages of development. The approach consisted of three key stages: pre-processing to remove duplicate data, feature selection using an adaptive artificial jelly optimisation algorithm (A2JO) to reduce complexity and prevent overfitting, and classification using a long short-term memory (LSTM) model to predict defective and non-defective code. Experiments were conducted on publicly available datasets, including Promise and NASA repositories, and the model was evaluated using metrics such as accuracy, F-measure, G-measure, and Matthews Correlation Coefficient (MCC). The results demonstrated high predictive performance, achieving accuracies of 93.41% and 92.8% for the Promise and NASA datasets, respectively. While the study highlighted the effectiveness of combining feature optimisation with LSTM-based prediction, its applicability may be influenced by dataset characteristics and the computational cost of model training. Nevertheless, it provides a valuable contribution to automated bug detection research by integrating deep learning with feature selection techniques.

**Khalid et al. (2023) Software Defect Prediction Analysis Using Machine Learning Techniques**

Khalid et al. (2023) investigated machine learning (ML) techniques for software defect prediction, focusing on improving model accuracy and precision on publicly available datasets. The study applied K-means clustering to categorise class labels and employed various classification models on selected features. To further enhance model performance, Particle Swarm Optimisation (PSO) was used to optimise the ML models. The models were evaluated using metrics including accuracy, precision, recall, F-measure, error metrics, and

confusion matrices. Results indicated that all ML and optimised ML models performed well, with Support Vector Machine (SVM) and optimised SVM achieving the highest accuracies of 99% and 99.80%, respectively. Other models, including Naive Bayes, Random Forest, and ensemble methods, also showed strong performance. While the study demonstrates the effectiveness of combining feature selection and model optimisation for defect prediction, its reliance on specific dataset characteristics may influence generalizability. Nonetheless, it contributes significantly to advancing automated bug detection techniques with high-accuracy ML approaches.

**Akhtar, N., Rana, A., Deshpande, P. P., Kumar, M., Parida, P. K., & Bajaj, K. K. (2023). Software bug prediction and detection using machine learning and deep learning. International Journal of Intelligent Systems and Applications in Engineering**

Akhtar et al. (2023) conducted a comprehensive study on the application of machine learning (ML) and deep learning (DL) techniques for software bug prediction and detection. The research focused on analysing data from code repositories, bug databases, and other software-related sources to identify patterns linking code attributes to defect occurrence. The study included a comparative evaluation of various ML and DL approaches, emphasising the importance of publicly accessible datasets and model interpretability. The authors highlighted the potential of hybrid methodologies that combine machine learning and deep learning to improve prediction accuracy and detection capabilities. While the paper provided a broad overview of existing techniques and their practical implications for software development, it also discussed current limitations and identified future research directions in automated bug detection and prediction.

**Shaon, M. S. H., & Akter, M. S. (2025). Modern Approaches to Software Vulnerability Detection: A Survey of Machine Learning, Deep Learning, and Large Language Models**

Shaon and Akter (2025) presented a comprehensive survey of modern approaches for automated software vulnerability detection, focusing on machine learning (ML), deep learning (DL), and large language model (LLM) techniques. The study analysed recent advances in feature representation, fine-tuning, generative methods, and prompt engineering, highlighting their ability to capture both syntactic and semantic aspects of source code. Key challenges, including limited real-world datasets, class imbalance, interpretability issues, and high computational costs, were critically discussed. The authors also outlined promising future directions, such as neuro-symbolic hybrid methods, parameter-efficient fine-tuning,

cross-language generalisation, continual learning, and explainable AI. By bridging the gap between classical feature-based methods and LLM-driven frameworks, the survey provides valuable insights for developing scalable, accurate, and interpretable vulnerability detection systems.

**Yadav, P. S., Rao, R. S., Mishra, A., & Gupta, M. (2024). Machine Learning-Based Methods for Code Smell Detection**

Yadav et al. (2024) conducted a comprehensive survey of machine learning (ML) techniques for code smell detection, which serve as early indicators of potential software quality issues. The study reviewed 42 relevant works from 2005 to 2024, covering a range of ML algorithms including Support Vector Machines, J48, Naive Bayes, and Random Forest, as well as traditional methods such as rule-based and Bayesian approaches. The authors highlighted challenges in code smell detection, including the lack of standardized definitions, difficulty in feature selection, and handling large-scale datasets. By evaluating multiple contributing factors and presenting class-wise distributions of ML algorithms, the study demonstrated the potential of ML methods to improve software design and development practices. The findings emphasize the practical value of ML in anticipating and addressing software design flaws, ultimately enhancing software quality and maintainability.

**Albattah, W., & Alzahrani, M. (2024). Software Defect Prediction Based on Machine Learning and Deep Learning Techniques**

Albattah and Alzahrani (2024) investigated machine learning (ML) and deep learning (DL) techniques for software defect prediction, emphasizing early-stage bug detection to enhance software reliability and reduce maintenance costs. The study evaluated eight widely used ML and DL algorithms using a large dataset compiled from five publicly available bug repositories, comprising around 60 software metrics such as cohesion, coupling, complexity, documentation, inheritance, and class size. Models were compared using performance metrics including accuracy, macro F1 score, weighted F1 score, and binary F1 score. Results indicated that the deep learning model, particularly LSTM, outperformed traditional ML algorithms, achieving an accuracy of 87%. The study highlights the effectiveness of combining extensive software metrics with deep learning approaches for early and accurate defect prediction, contributing to improved software quality and maintainability.

**Yousofvand, L., Soleimani, S., Rafe, V., & et al. (2026). Graph neural networks for precise bug localisation through structural program analysis**

Yousofvand et al. (2026) proposed a graph neural network (GNN)-based approach for precise bug localisation, addressing the challenge of identifying code segments responsible for program failures in increasingly complex software systems. The method represents source code as graphs encoding syntactic and semantic structures, labelling nodes using the Gumtree algorithm, and classifying them with a supervised GNN model into buggy or bug-free nodes. To handle class imbalance, the approach was evaluated using accuracy, precision, recall, and F1-score metrics. Experimental results demonstrated that the proposed method outperformed existing techniques, effectively localising a wide range of bug types, including undefined properties, functional bugs, variable naming errors, and variable misuse. This study highlights the potential of structural program analysis and graph-based deep learning models for automated, high-precision bug detection.

**Abdul Kadar, M. (2022). Automated code review and vulnerability detection using graph neural networks**

Abdul Kadar (2022) proposed a graph neural network (GNN)-based framework for automated code review and vulnerability detection, focusing on improving software security within modern development workflows, including DevSecOps. The approach represents source code as structural graphs to capture semantic relationships and extracts features for GNN-based classification of security vulnerabilities and code quality issues. The model achieved 93.7% accuracy across multiple programming languages, outperforming traditional static analysis tools by 27% and conventional deep learning approaches by 18%. When integrated into CI/CD pipelines, the system provided real-time feedback during code commits, reducing vulnerabilities by 76% and decreasing false positives by 41%. This study demonstrates the effectiveness of combining structural code representation with deep learning to enhance automated vulnerability detection and streamline code review processes.

**Yin, Y., Zhao, Y., Sun, Y., & Chen, C. (2023). Automatic Code Review by Learning the Structure Information of Code Graph. Sensors**

Yin et al. (2023) proposed an automated code review model that leverages structural information from code graphs to improve review efficiency. The study introduced the PDG2Seq algorithm, which converts program dependency graphs into unique sequences while preserving structural and semantic information. The model builds on the pre-trained

CodeBERT architecture, integrating both code sequence and structure information, and is fine-tuned for practical code review scenarios. Experimental results demonstrated significant improvements over baseline methods, as measured by BLEU, Levenshtein distance, and ROUGE-L metrics. This work highlights the potential of combining graph-based structural representations with deep learning models to enhance automated code review processes.

**Siachos, I., Kanakaris, N., & Karacapilidis, N. (2025). Software bug prediction using graph neural networks and graph-based text representations**

Siachos et al. (2025) proposed a hybrid approach for software bug prediction that combines graph-based text representations, word embeddings, and graph neural networks (GNNs) to leverage both structural and semantic information. Unlike prior methods that focus on individual components, the approach models textual data from issue tracking platforms as graphs and applies Graph Attention Networks (GATs) to predict software bugs. Experiments on four publicly available datasets from GitHub and Jira demonstrated improvements in accuracy, precision, and recall compared to existing graph-based machine learning models. This study underscores the potential of integrating textual information and graph-based learning for enhanced bug prediction in open-source software development environments.

**Viswanadhapalli, V. (2024). Automated bug detection and resolution using deep learning: A new paradigm in software engineering**

Viswanadhapalli (2024) presented an in-depth analysis of deep learning techniques for automated bug detection and resolution, highlighting their potential to improve software reliability and reduce debugging time. The study reviewed neural network architectures, including CNNs for token-based code analysis, RNNs and LSTMs for capturing sequential dependencies, and transformer-based models such as CodeBERT and GPT-4 for large-scale code understanding. The paper also discussed transfer learning and reinforcement learning approaches to enhance model adaptability and optimise corrective actions. While deep learning methods significantly improve accuracy and efficiency compared to traditional static and dynamic analysis, challenges remain, including the scarcity of high-quality labelled datasets, interpretability issues, and high computational costs. The study further proposed a hybrid deep learning approach combining multiple architectures to leverage their strengths and mitigate individual limitations, providing a promising direction for more effective automated bug detection and resolution in modern software engineering.

**Zymawy, H. (2025). Leveraging machine learning for automated code quality assessment and optimisation in modern software development**

Zymawy (2025) proposed a comprehensive machine learning-based framework for automated code quality assessment, optimisation, and intelligent software development workflows. The study employed transformer-based deep learning models trained on large-scale code repositories to perform automated code review, predictive bug detection, performance optimisation, and technical debt management. Experimental results demonstrated substantial improvements over traditional static analysis tools, including a 42% increase in bug detection accuracy, a 35% reduction in code review time, a 67% improvement in performance optimisation, and 89% accuracy in technical debt prediction. The framework was successfully deployed in production across multiple programming languages and large-scale codebases, highlighting the practical effectiveness of integrating advanced ML techniques into modern software engineering practices.

**Barrameda, R. B., & Ballera, M. (2025). Enhancing code quality: A CNN-based approach for readability classification and bug localisation in programming**

Barrameda and Ballera (2025) proposed a convolutional neural network (CNN)-based approach for automated code readability classification and bug localisation, aimed at improving programming education and software quality. The model employs a hybrid activation function combining ReLU and Leaky ReLU and processes structured code representations derived from lexical and syntactic analysis to extract hierarchical features indicative of code quality. Experiments on open-source datasets relevant to beginner computer science students achieved a classification accuracy of 82.4%. The study highlights the potential of deep learning to provide automated feedback, support scalable code evaluation, and enhance bug detection, while noting challenges such as overfitting and computational complexity.

### 2.2 Summary of Literature Review

| S/N | Author | Title | Summary | Limitations |
|---|---|---|---|---|
| 1 | Siva et al. (2023) | Automatic Software Bug Prediction Using Adaptive Artificial Jelly Optimisation With LSTM | Proposed a three-stage approach for software bug prediction: pre-processing, feature selection using adaptive artificial jelly optimisation (A2JO), and classification using | Dataset-specific performance; computational cost of LSTM and optimisation step; generalisation to unseen projects may be limited. |

| | | | | |
|---|---|---|---|---|
| | | | LSTM. Experiments on Promise and NASA datasets achieved accuracies of 93.41% and 92.8%, respectively. | |
| 2 | Khalid et al. (2023) | Software Defect Prediction Analysis Using Machine Learning Techniques | Investigated ML and optimised ML models for defect prediction using K-means for label categorisation and Particle Swarm Optimisation for model optimisation. SVM and optimised SVM achieved accuracies of 99% and 99.80%. | Reliance on specific dataset characteristics may not generalise well to different software contexts. |
| 3 | Akhtar et al. (2023) | Software Bug Prediction and Detection Using Machine Learning and Deep Learning | Reviewed ML and DL methods for bug prediction and detection from code repositories and bug databases, emphasising hybrid approaches that leverage multiple techniques for improved performance. | Broad survey; did not propose a specific novel predictive model. |
| 4 | Shaon & Akter (2025) | Modern Approaches to Software Vulnerability Detection: A Survey of ML, DL, and LLMs | Surveyed ML, DL, and LLM-based vulnerability detection, analysing feature representation, fine-tuning, generative methods, and prompt engineering. Highlighted challenges like dataset scarcity, class imbalance, and interpretability. | Focused on survey; practical implementation and evaluation of models were not presented. |
| 5 | Yadav et al. (2024) | Machine Learning-Based Methods for Code Smell Detection | Reviewed 42 studies on ML techniques for code smell detection, including SVM, Random Forest, J48, and Naive Bayes. Addressed challenges in feature selection, | It relies on small-scale datasets; generalisation to large industrial codebases is limited. |

| | | | | |
|---|---|---|---|---|
| | | | dataset scale, and lack of standardised definitions. | |
| 6 | Albattah & Alzahrani (2024) | Software Defect Prediction Based on Machine Learning and Deep Learning Techniques | Empirical study comparing 8 ML and DL algorithms using 5 public datasets with ~60 software metrics; LSTM outperformed others with 87% accuracy. | Computational cost of deep learning; performance may vary with different datasets. |
| 7 | Yousofvand et al. (2026) | Graph Neural Networks for Precise Bug Localisation | Proposed GNN-based bug localisation using graph representation of source code, node labelling via Gumtree, and supervised classification with evaluation on accuracy, precision, recall, and F1-score. | Dataset-dependent performance; class imbalance challenges; complexity of graph-based methods. |
| 8 | Abdul Kadar (2022) | Automated Code Review and Vulnerability Detection Using GNNs | Developed a GNN-based framework for automated code review and vulnerability detection with 93.7% accuracy, integrated into CI/CD pipelines to reduce vulnerabilities by 76%. | High computational cost; may require adaptation for specific programming languages or environments. |
| 9 | Yin et al. (2023) | Automatic Code Review by Learning the Structure Information of Code Graph | Proposed PDG2Seq algorithm to convert program dependency graphs into sequences; CodeBERT-based model integrates code sequence and structural info, improving BLEU, Levenshtein, and ROUGE-L metrics. | Focused on structure-sequence fusion; may require large datasets for fine-tuning. |
| 10 | Siachos et al. (2025) | Software Bug Prediction Using GNNs and Graph-Based Text Representations | Hybrid approach using GATs and graph-based text representations from issue tracking data; improved accuracy, precision, | Limited to textual issue data; generalisation to other datasets or programming languages may be |

| | | | | |
|---|---|---|---|---|
| | | | and recall on GitHub and Jira datasets. | limited. |
| 11 | Viswanadhapalli (2024) | Automated Bug Detection and Resolution Using Deep Learning | Reviewed DL architectures for bug detection (CNN, RNN, LSTM, transformers) and proposed hybrid models; discussed transfer learning and reinforcement learning for automated debugging. | High computational cost; interpretability of deep learning models; limited availability of high-quality labelled datasets. |
| 12 | Zymawy (2025) | Leveraging ML for Automated Code Quality Assessment and Optimisation | Proposed transformer-based DL framework for code review, bug detection, performance optimisation, and technical debt management; demonstrated 42% improvement in bug detection and 35% reduction in review time. | High resource requirements; may require extensive code repositories for training. |
| 13 | Barrameda & Ballera (2025) | Enhancing Code Quality: A CNN-Based Approach for Readability Classification and Bug Localisation | CNN-based approach with hybrid ReLU/Leaky ReLU activation for code readability classification and bug localisation; achieved 82.4% accuracy on student code datasets. | Focused on educational datasets; overfitting and computational complexity are challenges. |

### 2.3 Knowledge Gap/ Recommendation

While significant advances have been made in the development of machine learning and deep learning-based approaches for automated code review and bug detection, several important challenges remain unresolved. One key limitation lies in the insufficient integration of semantic and structural information from source code. Many existing methods rely primarily on sequential code representations or textual features, which may fail to capture the deeper relationships and dependencies within the program, limiting their ability to detect complex bugs and design flaws.

Additionally, although graph-based models and neural architectures such as Graph Neural Networks (GNNs) and transformers have shown promise, their high computational requirements and dependence on large, high-quality datasets restrict their practical applicability, especially in resource-constrained or real-time environments. Furthermore, many studies focus on a narrow set of programming languages or small-scale datasets, raising questions about the generalizability of the proposed approaches to diverse software projects and industrial-scale codebases.

Another gap is the limited research on combining different approaches to improve bug detection and code review. For example, few studies explore how to effectively use multiple types of information from the code, such as its structure, content, and runtime behaviour, together. While some work has tried to mix machine learning and deep learning methods, there is still a lack of systematic strategies for bringing these different sources of information together in a practical way.

Finally, although evaluation metrics such as accuracy, F1-score, and BLEU are commonly reported, there is a lack of standardised benchmarking frameworks for comparing different automated code review and bug detection methods. This inconsistency hampers the assessment of model robustness, scalability, and effectiveness in real-world software development pipelines.

These unresolved challenges highlight the need for more comprehensive, adaptable, and computationally efficient approaches that can effectively leverage structural, semantic, and behavioural aspects of code while supporting real-time deployment and cross-project generalisation.

## REFERENCES

1. Kavuri, S. (2025). AI-driven test automation frameworks: Enhancing efficiency and accuracy in software quality assurance. **International Journal of Applied Mathematics, 38**(10s), 699–710. https://doi.org/10.12732/ijam.v38i10s.990
2. Cheng, J. (2025). Research on improving the credibility and reliability of industrial Internet software testing quality assurance based on a digital twin. Journal of Technology Innovation and Engineering, 1(2). https://doi.org/10.63887/jtie.2025.1.2.2
3. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (2018). VulDeePecker: A deep learning-based system for vulnerability detection [Preprint]. Submitted January 5, 2018.

4. Siva, R., S K., Hariharan, B., & et al. (2023). Automatic software bug prediction using adaptive artificial jelly optimisation with long short-term memory. Wireless Personal Communications, 132, 1975–1998. https://doi.org/10.1007/s11277-023-10694-9
5. Fregnan, E., Petrulio, F., Di Geronimo, L., et al. (2022). What happens in my code reviews? An investigation into automatically classifying review changes. Empirical Software Engineering, 27, 89. https://doi.org/10.1007/s10664-021-10075-5
6. Fregnan, E., Petrulio, F., & Bacchelli, A. (2022). The evolution of the code during review: An investigation on review changes. Empirical Software Engineering**,** 27, 177. https://doi.org/10.1007/s10664-022-10205-7
7. Akhtar, N., Rana, A., Deshpande, P. P., Kumar, M., Parida, P. K., & Bajaj, K. K. (2023). Software bug prediction and detection using machine learning and deep learning. International Journal of Intelligent Systems and Applications in Engineering, 12(9s), 301–308. https://ijisae.org/index.php/IJISAE/article/view/4277
8. Shaon, M. S. H., & Akter, M. S. (2025). Modern Approaches to Software Vulnerability Detection: A Survey of Machine Learning, Deep Learning, and Large Language Models. Electronics, 14(22), 4449. https://doi.org/10.3390/electronics14224449
9. Meher, J. P., Biswas, S., & Mall, R. (2024). Deep learning-based software bug classification. Information and Software Technology, 166, 107350. https://doi.org/10.1016/j.infsof.2023.107350
10. Khalid, A., Badshah, G., Ayub, N., Shiraz, M., & Ghouse, M. (2023). Software Defect Prediction Analysis Using Machine Learning Techniques. Sustainability, 15(6), 5517. https://doi.org/10.3390/su15065517
11. Yadav, P. S., Rao, R. S., Mishra, A., & Gupta, M. (2024). Machine Learning-Based Methods for Code Smell Detection: A Survey. Applied Sciences, 14(14), 6149. https://doi.org/10.3390/app14146149
12. Albattah, W., & Alzahrani, M. (2024). Software Defect Prediction Based on Machine Learning and Deep Learning Techniques: An Empirical Approach. AI, 5(4), 1743-1758. https://doi.org/10.3390/ai5040086
13. Mienye, I. D., & Swart, T. G. (2024). A Comprehensive Review of Deep Learning: Architectures, Recent Advances, and Applications. Information, 15(12), 755. https://doi.org/10.3390/info15120755
14. Yousofvand, L., Soleimani, S., Rafe, V., & et al. (2026). Graph neural networks for precise bug localisation through structural program analysis. Automated Software Engineering, 33, 17. https://doi.org/10.1007/s10515-025-00556-y

15. Shi, Y., Yin, Y., Yu, M., & Chu, L. (2024). CogCol: Code Graph-Based Contrastive Learning Model for Code Summarisation. Electronics, 13(10), 1816. https://doi.org/10.3390/electronics13101816
16. Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2018). code2vec: Learning distributed representations of code. arXiv. https://doi.org/10.48550/arXiv.1803.09473
17. Abdul Kadar, M. (2022). Automated code review and vulnerability detection using graph neural networks: Enhancing DevSecOps workflows. World Journal of Advanced Engineering Technology and Sciences, 5(1), 113–122. https://doi.org/10.30574/wjaets.2022.5.1.0031
18. Yin, Y., Zhao, Y., Sun, Y., & Chen, C. (2023). Automatic Code Review by Learning the Structure Information of Code Graph. Sensors, 23(5), 2551. https://doi.org/10.3390/s23052551
19. Cui, M., Long, S., Jiang, Y., & Na, X. (2022). Research on Software Defect Prediction Model Based on Complex Network and Graph Neural Network. Entropy, 24(10), 1373. https://doi.org/10.3390/e24101373
20. Siachos, I., Kanakaris, N., & Karacapilidis, N. (2025). Software bug prediction using graph neural networks and graph-based text representations. Expert Systems with Applications, 240, 125290. https://doi.org/10.1016/j.eswa.2024.125290
21. Morasca, S., & Lavazza, L. (2020). On the assessment of software defect prediction models via ROC curves. Empirical Software Engineering**,** **25**, 3977–4019. https://doi.org/10.1007/s10664-020-09861-4
22. Golovnev, N., Starovoytov, N., & Staroletov, S. (2025, June). Challenges in automating error-fixing commit classification for Linux Kernel and cyber-physical systems. In 2025 IEEE 26th International Conference of Young Professionals in Electron Devices and Materials (EDM). https://doi.org/10.1109/EDM65517.2025.11096858
23. Viswanadhapalli, V. (2024). Automated bug detection and resolution using deep learning: A new paradigm in software engineering. International Journal of Engineering and Computer Science, 13. https://doi.org/10.18535/ijecs/v13i04.4816
24. Zymawy, H. (2025). Leveraging machine learning for automated code quality assessment and optimisation in modern software development: A comprehensive framework for intelligent software engineering (Report No. 007). Goldsmiths University of London. https://doi.org/10.13140/RG.2.2.16288.24320

25. Barrameda, R. B., & Ballera, M. (2025). Enhancing code quality: A CNN-based approach for readability classification and bug localisation in programming. In Computer and Electrical Engineering. https://doi.org/10.3233/ATDE250732