Check for updates

Concurrency and Computation: Practice and Experience

WILEY

RESEARCH ARTICLE OPEN ACCESS

Deep Learning Architectures for Software Fault Prediction: The Impact of Error-Type Metrics and Class Imbalance

Khoa Phung[1] | Mehmet E. Aydin[1,2] | Emmanuel Ogunshile[1]

[1]School of Computing and Creative Technologies, University of the West of England, Bristol, UK | [2]Department of Industrial Engineering, Istanbul Commerce University, Istanbul, Turkiye

Correspondence: Khoa Phung (khoa.phung@uwe.ac.uk)

Received: 30 December 2024 | **Revised:** 10 November 2025 | **Accepted:** 18 November 2025

Keywords: class imbalance | deep learning | error-type metrics | graph neural networks | software fault prediction

ABSTRACT

Software fault prediction (SFP) plays a crucial role in modern software development by enabling early identification of fault-prone modules and efficient allocation of testing resources. While deep learning approaches have shown promise in this domain, challenges persist regarding architectural choices, metric selection, and class imbalance issues. This study presents a comprehensive comparison between deep neural networks (DNNs) and hybrid Graph Neural Network-Long Short-Term Memory (GNN+LSTM) models for SFP, investigating their effectiveness when combined with both conventional software metrics and error-type metrics. We evaluate these approaches on four real-world Java projects: ANTLR v4, JUnit, OrientDB, and Elastic Search. Our results demonstrate that GNN+LSTM models consistently outperform traditional DNN approaches, achieving improvements of up to 4% in accuracy and 4% in F1-score. However, we identify challenges in combining different metric sets, with performance actually degrading compared to our previous study using error-type metrics alone, suggesting potential multi-collinearity issues. Additionally, we examine the effectiveness of the synthetic minority oversampling technique (SMOTE) in addressing the class imbalance issue, observing improvements of up to 6.6% in accuracy for GNN+LSTM models in severely imbalanced datasets. Our findings provide practical insights for selecting appropriate model architectures and metric combinations in SFP while highlighting the importance of carefully considering feature interactions and class imbalance mitigation strategies.

1 | Introduction

Software fault prediction (SFP) has become increasingly critical in modern software development, where the cost and impact of software failures continue to escalate. Early identification of fault-prone modules not only enables more efficient allocation of testing resources but also significantly reduces maintenance costs and enhances software reliability. Traditional SFP approaches have relied heavily on machine learning models trained with conventional software metrics, such as the Chidamber and Kemerer (CK) suite [1], which quantify various aspects of software quality including coupling, cohesion, and inheritance. However, these conventional approaches have encountered significant limitations, particularly a "performance ceiling" where additional computational complexity yields diminishing returns. This ceiling stems from three primary challenges: insufficient dataset quality, limited metric representativeness, and inability to capture intricate relationships within software systems [2].

The emergence of deep learning techniques has opened new avenues for addressing these fundamental limitations. Deep neural networks (DNNs) are particularly well-suited for SFP due to

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2025 The Author(s). *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

their ability to learn hierarchical feature representations and capture complex non-linear patterns in high-dimensional software metrics data [3]. Graph neural networks (GNNs) address the limitation that software systems inherently possess graph-like structures through dependencies including call graphs and module interdependencies, which traditional approaches fail to model effectively [4]. Long short-term memory (LSTM) networks capture the temporal evolution of software metrics across development cycles, as software development is inherently a temporal process where fault patterns exhibit sequential dependencies [5]. Various architectures have shown promise in this domain: Convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs) have achieved enhanced precision in defect identification [6], while hybrid approaches combining GNNs [4] with LSTM networks [7] have successfully captured both structural and temporal dependencies in software systems [8]. Despite these advances, several critical challenges remain unaddressed in maximising the potential of deep learning for SFP.

A fundamental challenge in SFP is the inherent class imbalance in software fault datasets, where defective modules typically represent a small minority of cases. This imbalance can severely impact model performance, often resulting in biased predictions favouring the majority class (non-faulty modules). While techniques such as the synthetic minority oversampling technique (SMOTE) [9] have been employed to address this issue, the effectiveness of these approaches remains limited by the underlying quality and representativeness of the metrics used for prediction. Traditional software metrics, including CK and Halstead metrics [10], often fail to capture the subtle characteristics that distinguish fault-prone modules, particularly in modern, complex software systems.

To address these limitations, previous research introduced error-type metrics [11], a set of application-independent metrics derived from Java runtime error patterns. These metrics provide a more nuanced representation of potential fault indicators, capturing information that conventional metrics often miss. When combined with DNN models, error-type metrics demonstrated substantial improvements over traditional approaches, achieving up to 40% gains in area under the curve (AUC) and receiver operating characteristic (ROC) scores [12]. These promising results suggest that the combination of sophisticated metrics and advanced deep learning architectures could break through the aforementioned performance ceiling.

Building upon these findings, the current study aims to conduct a comprehensive comparative analysis between DNN and hybrid GNN-LSTM models in SFP. Our investigation focuses on three key aspects:

1. A systematic comparison between DNN architectures and hybrid GNN-LSTM models to determine their relative effectiveness in SFP. This comparison aims to understand which approach better captures the inherent patterns in software systems that lead to faults, particularly considering the structural and temporal characteristics of software development.
2. An investigation of how error-type metrics, when combined with conventional software metrics, influence the performance of both DNN and GNN-LSTM models. This analysis seeks to understand whether different architectural approaches respond differently to these enhanced metrics and whether the graph-based representation of GNNs provides additional benefits when processing error-type metrics.
3. An examination of how the application of SMOTE affects the performance of different deep learning architectures in handling class imbalance. This involves comparing the prediction performance of both DNN and GNN+LSTM models on datasets with and without SMOTE application, considering how each architecture responds to synthetic minority samples.

To guide this investigation, we address the following research questions:

- RQ1: How do DNN models compare with hybrid GNN+LSTM models in SFP when trained on error-type metrics?
- RQ2: How does the integration of error-type metrics with conventional software metrics affect the prediction performance of DNN models versus GNN+LSTM models in identifying fault-prone software modules?
- RQ3: To what extent does the application of SMOTE improve the performance of DNN and GNN+LSTM models in handling imbalanced fault prediction datasets?

This study contributes to the field by providing the first systematic comparison between DNN and GNN+LSTM architectures in SFP, demonstrating the effectiveness of hybrid models and revealing unexpected challenges in combining different metric sets. Our experiments on four real-world Java projects show that GNN+LSTM models consistently outperform traditional DNNs with improvements of up to 4% in accuracy and F1-score, while SMOTE proves effective in addressing class imbalance with accuracy gains reaching 6.6% in severely imbalanced datasets. However, we identify that combining error-type metrics with conventional software metrics may introduce multi-collinearity issues that can degrade performance compared to using error-type metrics alone.

The significance of this research lies in its potential to identify the most effective deep learning architecture for SFP while leveraging novel metrics. By conducting a direct comparison between DNN and GNN+LSTM approaches, this work provides practical insights for choosing the most appropriate model architecture for software quality assurance processes, particularly considering the structural and temporal characteristics inherent in software systems.

The remainder of this paper is organised as follows: Section 2 presents a comprehensive review of related work, covering software metrics and deep learning applications in SFP. Section 3 details our methodology, including data preprocessing approaches, model architectures, and evaluation measures. Section 4 presents our experimental results, comparison with the state-of-the-art, and discussions on the limitations. Finally, Section 5 concludes the paper and outlines promising directions for future research.

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

2 | Background and Related Work

This section analyses current research in SFP, focusing on the landscape of software metrics and the growing application of deep learning techniques.

2.1 | Software Metrics

Software metrics, as defined by Goodman [13], represent different properties of a software product that are continually monitored throughout the development lifecycle. These properties, including aspects like coupling, cohesion, and inheritance, each correspond to a specific metric. Such metrics are crucial for determining the external quality of software modules in terms of factors like testability, reliability, and fault-proneness [14].

Software metrics can be classified into several categories based on their measurement focus. Product metrics measure the characteristics of the software product itself. These include size metrics such as function points (FP), Source Lines of Code (SLOC), Number of Classes, and Number of Methods [15]. Complexity metrics encompass McCabe's cyclomatic complexity, weighted methods per class (WMC), and response for a class (RFC) [16]. Quality metrics include measurements such as Defects per FP, Defects per KSLOC, and the Maintainability Index [17]. Additionally, process metrics focus on the software development process itself, examining factors such as development effort, testing effort, number of revisions, and number of developers involved in the project [18].

As highlighted by Rathore and Kumar [19], the effectiveness of metrics can vary significantly based on the development environment (open-source vs. commercial), application domain (embedded systems, web applications, etc.), programming language, and development methodology.

Recent research has shown that combining different types of metrics can improve fault prediction accuracy [20]. This finding has led to the development of hybrid approaches that integrate traditional code metrics, object-oriented metrics, process metrics, and domain-specific metrics. Addressing the need for domain-independent metrics in this context, in our previous studies [21] and [11], the Stream X-Machine approach [22] has been used to model and generate test cases for Java runtime errors, extract error-type data (ESM values), and create domain-independent metrics.

2.2 | Deep Learning in SFP

Deep neural networks (DNNs) have emerged as a transformative technology for SFP, demonstrating superior capabilities in learning complex patterns and high-dimensional data representations. This section provides a comprehensive review of DNN-related advancements in SFP, with a focus on recent studies and state-of-the-art methodologies.

The initial adoption of DNNs in SFP focused on leveraging their capacity to handle large feature spaces and non-linear relationships. One of the earlier contributions by Kumudha and Venkatesan [23] employed radial basis function neural networks (RBFNNs) optimised with adaptive dimensional bio-geography based optimisation (ADBBO). The approach outperformed conventional machine learning techniques on NASA datasets, showcasing the potential of combining neural network architectures with meta-heuristic algorithms. Similarly, Yang et al. [24] introduced the Deeper model, which utilised a deep belief network (DBN) to derive high-level features from raw change metrics. Evaluations on six open-source projects revealed that Deeper outperformed logistic regression and other traditional methods by up to 32% in bug detection rates.

Hybrid models that combine DNNs with other architectures have been widely adopted to enhance SFP. For instance, Manjula and Florence [25] integrated genetic algorithms (GA) with DNNs for feature selection and classification. Their approach, tested on PROMISE datasets, achieved unprecedented accuracy rates (e.g., 98% on the PC4 dataset), highlighting the effectiveness of hybrid optimisation techniques. Another notable contribution was made by Jouybari et al. [8], who demonstrated the efficacy of combining graph neural networks (GNNs) and long short-term memory (LSTM) networks. By effectively capturing the sequential and graph-based nature of software data, this hybrid architecture significantly improved defect prediction, particularly in cross-project scenarios.

Attention mechanisms have revolutionised deep learning by enabling models to focus on critical features while ignoring irrelevant data. Yu et al. [26] introduced the defect prediction via the self-attention mechanism (DPSAM) model, which leveraged self-attention layers to extract semantic relationships between software components. The model achieved an average F1-score of 70% on seven PROMISE datasets, outperforming traditional machine learning and earlier deep learning techniques. Building on this, transformer architectures have demonstrated significant potential in SFP tasks. Recent studies have adapted transformers to defect prediction, with results showing state-of-the-art performance on both within-project and cross-project tasks [27, 28]. These architectures excel at modelling long-range dependencies and hierarchical relationships in software metrics, making them particularly effective for SFP.

Ensemble learning approaches, such as stacking and boosting, have been combined with DNNs to further enhance predictive performance. For example, Albahli [29] and Chen et al. [30] demonstrated that ensemble models integrating DNNs with Decision Tree, Random Forest, or XGBoost significantly improved prediction accuracy across public datasets such as NASA and PROMISE repositories. Additionally, transfer learning techniques have gained traction in SFP, allowing models trained on one domain to be fine-tuned for another. Pre-trained models leveraging large-scale software repositories have shown remarkable generalisation capabilities, particularly in cross-project scenarios [31, 32].

Class imbalance remains a critical challenge in SFP, as fault-prone modules often constitute a minority in datasets. Techniques such as the synthetic minority oversampling technique (SMOTE) have been employed to mitigate this issue. Gamel et al. [33] demonstrated that SMOTE effectively improved the performance of DNNs in predicting minority classes. Recent

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

advancements include cost-sensitive learning frameworks that adapt loss functions to prioritise fault-prone modules. These frameworks ensure balanced predictions by penalising misclassification of minority classes, thereby enhancing overall model performance.

2.3 | Graph Neural Networks

Graph neural networks (GNNs) and long short-term memory (LSTM) networks have emerged as complementary technologies for SFP, with GNNs excelling at capturing structural relationships in software systems and LSTMs providing temporal pattern recognition capabilities. Their combination offers a comprehensive approach to understanding both the architectural dependencies and evolutionary patterns in software development.

Graph neural networks have demonstrated significant effectiveness in SFP due to their ability to model complex relationships in software systems. Unlike traditional neural networks, GNNs effectively capture the interconnected nature of software components by propagating information through dependency graphs. The core computational process in GNNs can be summarised as:

$$h_i^{(k+1)} = \sigma\left(W^{(k)} \cdot \mathrm{AGG}\left(\{h_j^{(k)} | j \in \mathcal{N}(i)\}\right) + b^{(k)}\right), \quad (1)$$

where $h_i^{(k)}$ represents the feature vector of node i at the k-th layer, $\mathcal{N}(i)$ denotes the set of neighbouring nodes, AGG$(\cdot)$ is an aggregation function, and σ represents an activation function.

GNNs have demonstrated remarkable effectiveness in capturing software dependencies, largely because they can automatically learn complicated hidden patterns from high-dimensional data [34]. Typically, these patterns are derived from representations such as abstract syntax trees (ASTs) [4]. Existing deep learning approaches for extracting AST-based features either compute node attributes from the leaves up to the root in a bottom-up manner [35], employ specially designed relationships between AST nodes [36], or focus on only a part of the AST to reduce memory overhead [37]. However, each of these approaches has certain limitations. Bottom-up computation can struggle to capture long-range dependencies between distant nodes [35]. Similarly, defining additional relationships between nodes in an AST beyond its native structure can introduce complexity without clearly contributing new insights on source-code defectiveness [36]. Furthermore, restricting the analysis to only a portion of the AST, as implemented by some methods [37], may be memory efficient but can omit valuable structural information about the software module.

More recently, researchers have demonstrated that GNN-extracted features can also be beneficial for traditional classification models [38]. In particular, they combined features derived from a control flow graph (CFG) and AST-based local representations (extracted by a Tree-Based Convolutional Neural Network) with contextual vectors from an attention-based GNN. This process yields a unified feature vector for a given source code fragment. Nonetheless, to enforce uniform dimensionality for code components of varying sizes, they limited the size of both GNN layers and the feature vectors themselves, which can result in the loss of potentially critical information.

However, recent research has revealed significant limitations in applying GNNs alone to software fault prediction. Cui et al. [39] identified critical shortcomings of pure GNN approaches, noting that "the software defect prediction model based on the complex network only considers the graph structure of the software, ignoring the properties of the software module itself." Their empirical analysis demonstrated that GNN-only methods fail to effectively integrate node-level features (such as code metrics and historical fault information) with structural graph information, resulting in incomplete representation learning. The study showed that models combining both structural and node-level information significantly outperformed plain GNN approaches across multiple evaluation metrics, highlighting the necessity for hybrid architectures that can leverage both types of information simultaneously.

2.4 | Long Short-Term Memory Networks

Complementing the structural analysis capabilities of GNNs, long short-term memory (LSTM) networks were proposed by Hochreiter and Schmidhuber as a specialised form of recurrent neural network (RNN) designed to handle long-distance dependencies in sequential data [40]. Since their inception, numerous variants and topologies have emerged [41, 42], reflecting the growing interest in LSTM-based solutions for challenging deep learning tasks. In document classification, for instance, hierarchical attention networks built on LSTMs proved effective in extracting contextual dependencies [43], while a lexicon-enhanced LSTM architecture was shown to improve word representations in sentiment analysis [44].

The primary mechanism that distinguishes LSTM from traditional RNNs lies in its gating structure, which helps overcome issues such as the vanishing gradient problem. Specifically, an LSTM cell incorporates three main gates to regulate the flow of information across time steps:

$$f_t = \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right), \quad (2)$$

$$i_t = \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right), \quad (3)$$

$$\tilde{C}_t = \tanh\left(W_c x_t + U_c h_{t-1} + b_c\right), \quad (4)$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t, \quad (5)$$

$$o_t = \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right), \quad (6)$$

$$h_t = o_t \odot \tanh(C_t), \quad (7)$$

where f_t, i_t, and o_t are the forget, input, and output gates, respectively; C_t is the cell state, and h_t is the hidden state. By adjusting these gates, LSTMs effectively capture both short-term and long-term temporal patterns, making it possible to address various sequence learning challenges [42].

In SFP, recent studies have capitalised on these properties to model the temporal dynamics of source code changes and defect introduction. For example, Lin et al. [45] employed a Bi-LSTM network to learn cross-project transfer representations, thereby enhancing vulnerability detection for unfamiliar code. Within-project defect prediction has also benefited from LSTM-based approaches, as demonstrated in the work of Fan et al. [46] and Liang et al. [5], both of whom leveraged LSTM

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

to extract feature representations from historical software data. Beyond purely recurrent strategies, a deep tree-based model was proposed by [35], illustrating that hybrid architectures can further refine LSTM-derived features for defect prediction.

In summary, the comprehensive review of existing literature reveals several critical gaps that motivate the proposed GNN+LSTM architecture in this study:

1. Traditional DNN approaches treat software metrics as independent features, failing to capture the complex interdependencies inherent in software systems [2, 6], particularly evident in modern software architectures where component relationships significantly influence fault propagation patterns.
2. Pure GNN approaches suffer from significant limitations as identified by Cui et al. [39], focusing exclusively on graph structure while ignoring crucial node-level properties such as specific code metrics, historical fault information, and semantic features, resulting in incomplete representation learning.
3. Most existing approaches rely on conventional software metrics from classical datasets (NASA and PROMISE repositories) that have been subject to data quality and consistency concerns [47, 48], while conventional metrics often fail to capture subtle error-prone characteristics in contemporary software systems.
4. Although some studies have explored combining structural and temporal features [49, 50], most approaches restrict themselves to conventional software metrics without considering domain-independent, error-specific characteristics.

To address these gaps, this study proposes a hybrid GNN+LSTM architecture that combines structural dependency modelling capabilities with temporal pattern recognition strengths. The GNN component captures interconnected software component relationships through dependency graphs, while the LSTM component processes sequential patterns in metric evolution. Furthermore, this work applies error-type metrics, previously introduced in our earlier studies [11, 21], to investigate their effectiveness when combined with advanced deep learning architectures, providing a systematic approach to bridge current SFP methodological limitations.

3 | Methodology

This section presents our research methodology, detailing the software fault datasets, software metric selection, deep learning architectures, evaluation measures, and data preprocessing techniques employed in our study.

3.1 | Software Fault Datasets

Our study utilises software fault datasets derived from the BugHunter Dataset [51]. The BugHunter Dataset was chosen over other public repositories such as NASA and PROMISE due to its focus on faulty and fixed states of source code, providing a more accurate representation of software faults across project lifespans [51]. Unlike traditional repositories that may contain erroneous or superfluous information [47], BugHunter's approach captures the dynamic nature of software metrics during bug resolution.

We selected four actively maintained, open-source Java projects for our analysis: ANTLR v4, JUnit, OrientDB, and Elastic Search. These projects were chosen based on their diverse characteristics, active maintenance status, and substantial user bases, ensuring the generalisability of our findings. Table 1 provides a detailed overview of these projects.

For each project, we constructed two distinct datasets to examine the impact of error-type metrics on deep learning model performance:

- **Software Metrics Dataset:** Following our previous work [52], this baseline dataset contains conventional software metrics measured exclusively at the class level. These metrics have been extracted from the BugHunter Dataset and independently validated using Metrics Reloaded [53], ensuring data accuracy and reliability. They provide comprehensive measurements of various software characteristics including size, complexity, and object-oriented properties.
- **Full Dataset:** This enhanced dataset includes both conventional software metrics and error-type metrics. The error-type metrics comprise four specific Java Runtime Exception (JRE) measurements: ESM IndexOutOfBounds, ESM NullPointer, ESM ClassCast, and ESM Arithmetic. This combination allows us to evaluate how the addition of error-type information affects prediction accuracy.

In both datasets, the dependent variable is the `Fault Proneness` of each software module. This dual-dataset approach enables us to systematically evaluate the contribution of error-type metrics to fault prediction accuracy across different DNN-based models.

To illustrate the structure and content of our datasets, Tables 2–5 present sample records from each project's Full Dataset. These examples demonstrate how error-type metrics are combined with conventional software metrics to create a comprehensive profile of each software module. For instance, the JUnit project dataset includes both the ESM values quantifying exception occurrences and traditional metrics such as Lines of Code (LOC), Lack of Cohesion in Methods 5 (LCOM5), and Weighted Methods per Class (WMC), providing a comprehensive view of each module's characteristics and its fault history.

All experimental evaluations were conducted using both datasets for each project, allowing us to quantify the specific impact of error-type metrics on model performance. This comparative approach provides insights into whether and to what extent the inclusion of error-type metrics enhances the predictive capabilities of different deep learning architectures.

For reproducibility and further research, we have made all datasets publicly available as CSV files through our GitHub

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 | Selected projects and their descriptions.

Project name	No. software modules	Forks	kLOC	Description
ANTLR v4	469	1559	68	ANTLR v4 is a widely used parser generator in language processing. It allows developers to read, process, execute, or translate structured text and binary files. It is known for its versatility in compiler construction and domain-specific language creation
JUnit	351	2826	43	JUnit is a popular Java framework for writing and executing unit tests. It plays a crucial role in test-driven development (TDD) and ensures software reliability by providing simple annotations and assertions for testing
OrientDB	5514	792	621	OrientDB is a highly scalable NoSQL database that combines document and graph database features. It is widely used for its speed and flexibility in handling complex relationships within large datasets
Elastic Search	18,324	14,303	995	Elastic Search is a widely adopted RESTful search engine built on Apache Lucene. It is used for distributed search and analytics, offering high-speed indexing and querying capabilities for large volumes of data

TABLE 2 | Example of the ANTLR v4 dataset.

Modules	ESM IndexOutOfBounds	ESM NullPointer	ESM ClassCast	ESM arithmetic	LOC	LCOM5	WMC	Fault proneness
1	1	208	33	3	565	2	78	Yes
2	1	57	13	0	205	5	52	Yes
3	8	260	18	0	833	10	161	No
4	189	593	42	18	1067	3	152	Yes

TABLE 3 | Example of the JUnit dataset.

Modules	ESM IndexOutOfBounds	ESM NullPointer	ESM ClassCast	ESM arithmetic	LOC	LCOM5	WMC	Fault proneness
1	0	9	0	0	40	5	5	No
2	0	3	0	0	72	1	8	Yes
3	4	29	0	0	166	14	23	Yes
4	1	37	5	0	101	3	8	Yes

TABLE 4 | Example of the OrientDB dataset.

Modules	ESM IndexOutOfBounds	ESM NullPointer	ESM ClassCast	ESM arithmetic	LOC	LCOM5	WMC	Fault proneness
1	0	8	1	0	38	1	8	No
2	8	209	14	0	583	3	135	Yes
3	5	226	82	1	638	5	176	Yes
4	2	1428	38	7	1158	1	226	Yes

TABLE 5 | Example of the elastic search dataset.

Modules	ESM IndexOutOfBounds	ESM NullPointer	ESM ClassCast	ESM arithmetic	LOC	LCOM5	WMC	Fault proneness
1	32	263	6	1	161	1	17	Yes
2	70	139	51	60	351	3	18	Yes
3	0	61	0	0	71	1	4	Yes
4	32	77	1	6	72	1	16	No

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

repository.[1] Both the Software Metrics Dataset and Full Dataset versions are provided for each project, enabling other researchers to replicate our experiments and conduct additional analyses.

3.2 | Deep Learning Architectures

Our study investigates and compares two distinct deep learning approaches: standalone deep neural networks (DNNs) and hybrid graph neural network with long short-term memory (GNN+LSTM) models. These architectures are evaluated on both the Full Dataset and the Conventional Software Metrics Dataset, with model sizes adapted according to dataset complexity.

3.2.1 | Model Architecture Strategy

To accommodate varying dataset sizes across our selected projects, we configure these models in three categories based on the number of software modules (instances) in each dataset. As shown in Table 1, our projects exhibit substantial variation in dataset size, ranging from 351 software modules in JUnit to 18,324 modules in Elastic Search. This variation necessitates a systematic approach to model sizing that balances computational efficiency with predictive capacity.

Our model selection strategy follows established practices in deep learning architecture design [54] and is defined as follows:

- **Small Models:** Designed for datasets with fewer than 500 software modules. These architectures prevent overfitting in limited data scenarios while maintaining sufficient model capacity for effective feature learning. This category applies to ANTLR v4 (469 modules) and JUnit (351 modules).
- **Medium Models:** Suitable for datasets with 500 to 10,000 software modules. These configurations provide increased capacity for moderate-sized datasets, balancing model complexity with available training data. This category applies to OrientDB (5514 modules).
- **Large Models:** Designed for datasets with more than 10,000 software modules. These architectures utilise maximum model capacity to leverage extensive data for complex pattern recognition and feature interaction modelling. This category applies to Elastic Search (18,324 modules).

This systematic scaling approach ensures that model complexity appropriately matches dataset characteristics, following the principle that model capacity should be proportional to the amount of available training data to avoid both underfitting and overfitting [55].

3.2.2 | DNN Architecture

We implement three DNN variants (Small, Medium, and Large) corresponding to the dataset size categories outlined above. Each variant employs batch normalisation to stabilise training [56], dropout layers (rate = 0.5) to prevent overfitting [57], and LeakyReLU activation functions to address the dying ReLU problem [58].

- Small DNN: Three dense layers (64 → 32 → 1 neurons)
- Medium DNN: Four dense layers (128 → 64 → 32 → 1 neurons)
- Large DNN: Five dense layers (256 → 128 → 64 → 32 → 1 neurons)

This progressive scaling approach allows the networks to adapt to varying levels of data complexity while maintaining computational efficiency. The smaller architectures help prevent overfitting in limited datasets, while larger networks can capture more complex feature interactions in extensive datasets.

3.2.3 | GNN+LSTM Architecture

Our hybrid architecture systematically integrates graph attention networks (GAT) [59] with LSTM networks to address the fundamental limitations of applying either approach in isolation to software fault prediction. This integration is motivated by the complementary nature of these architectures: GNNs excel at capturing non-sequential, structural dependencies within software systems, while LSTMs specialise in modelling sequential patterns and temporal dynamics that characterise software evolution and fault development processes [60].

3.2.3.1 | Architectural Integration and Data Flow. The hybrid architecture implements a sequential processing pipeline that systematically combines structural and temporal analysis capabilities. Figure 1 illustrates the complete data flow and architectural integration process. The data flow proceeds through three distinct stages:

Stage 1: Graph-based Structural Analysis. The input software metrics are first organised as nodes in a dependency graph, where each software module represents a node and edges encode relationships such as method calls, inheritance hierarchies, and coupling dependencies. The GAT layers process this graph structure using multi-head attention mechanisms [59], computing node representations that capture both local neighbourhood information and global structural patterns. The attention mechanism enables the model to differentially weight the importance of various structural dependencies, addressing the limitation identified by Cui et al. [39] where traditional graph methods fail to effectively integrate node-level properties with structural information.

The GAT computation for each node i at layer l follows:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l)} \right) \tag{8}$$

where $\alpha_{ij}^{(l)}$ represents the attention coefficient between nodes i and j, computed through a learnable attention mechanism that identifies the most relevant structural relationships for fault prediction.

Stage 2: Temporal Sequence Preparation. The learned node representations from the GAT layers are aggregated using global max pooling [61], which preserves the most salient structural features while creating fixed-dimensional representations suitable

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

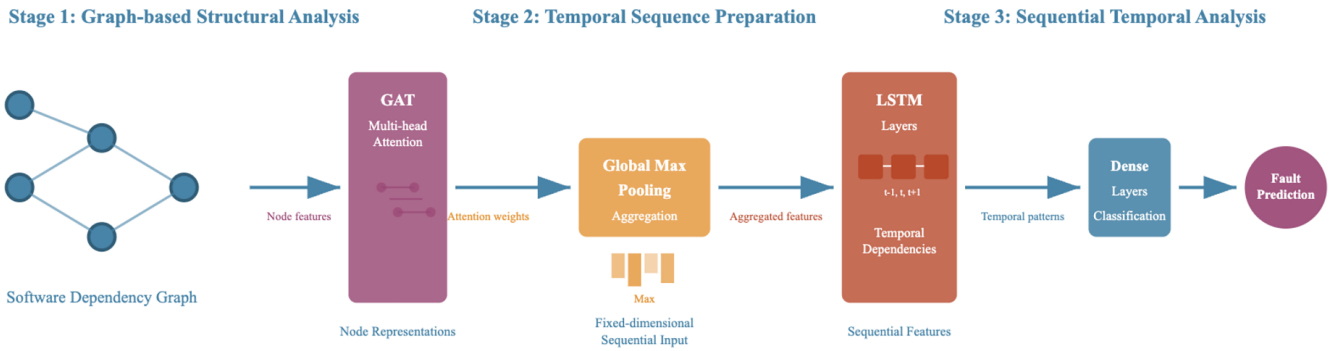


FIGURE 1 | GNN+LSTM hybrid architecture for software fault prediction.

for sequential processing. This aggregation step is crucial as it transforms variable-sized graph representations into consistent sequential inputs that can be effectively processed by LSTM layers, as shown in Figure 1. The pooling operation ensures that critical structural information—such as highly coupled modules or complex inheritance patterns—is retained in the sequential representation.

Stage 3: Sequential Temporal Analysis. The aggregated structural features are then processed by LSTM layers, which model the temporal evolution of these structural patterns. This is particularly important because software faults often emerge through temporal processes that cannot be captured by static structural analysis alone [60]. For instance, gradual increases in coupling complexity, evolving error patterns, or the accumulation of technical debt represent temporal dynamics that influence fault probability over time.

The LSTM processing captures these temporal dependencies through its gating mechanisms:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{9}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{10}$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{11}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{12}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{13}$$

$$h_t = o_t * \tanh(C_t) \tag{14}$$

where x_t represents the structural features at time step t, enabling the model to learn how structural patterns evolve and contribute to fault introduction.

3.2.3.2 | Rationale for Hybrid Integration. The combination of GAT and LSTM addresses three fundamental limitations of existing approaches. First, traditional DNNs treat software metrics as independent features, missing the complex interdependencies that characterise modern software systems. The GAT component directly addresses this by explicitly modelling structural relationships through graph attention mechanisms.

Second, pure GNN approaches, while effective at capturing structural dependencies, fail to model the temporal dynamics of software evolution. As noted by Meng et al. [60], software defects often result from sequential processes—such as evolving complexity patterns, changing development practices, or the gradual introduction of error-prone patterns—that require temporal modelling capabilities. The LSTM component provides this temporal understanding by processing the learned structural representations as sequential data.

Third, pure LSTM approaches, while capable of temporal modelling, lack the structural awareness necessary to understand how different software components interact and influence each other. The GAT component provides this structural foundation, ensuring that the temporal analysis is grounded in accurate representations of software architecture and dependencies.

3.2.3.3 | Implementation Specifications. The architecture is implemented in three scales corresponding to dataset complexity:

Small GNN+LSTM:

- GAT: Single layer with 64 hidden units and 4 attention heads
- Global max pooling aggregation
- LSTM: Two layers with 64-dimensional hidden states
- Dense layers: 32 → 1 neurons

Medium GNN+LSTM:

- GAT: Two layers with 128 hidden units each and 4 attention heads
- Global max pooling aggregation
- LSTM: Two layers with 128-dimensional hidden states
- Dense layers: 64 → 1 neurons

Large GNN+LSTM:

- GAT: Three layers with 256 hidden units each and 4 attention heads
- Global max pooling aggregation
- LSTM: Two layers with 256-dimensional hidden states
- Dense layers: 128 → 1 neurons

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Each variant incorporates batch normalisation for training stability, dropout (0.5) for regularisation, and LeakyReLU activations to address gradient flow issues. The multi-head attention mechanism in GAT layers (4 heads across all variants) provides diverse perspectives on structural relationships, while the two-layer LSTM configuration balances temporal modelling capacity with computational efficiency.

This hybrid architecture represents a systematic solution to the identified limitations of existing approaches, providing comprehensive fault prediction capabilities that leverage both structural dependencies and temporal evolution patterns. The integration enables the model to understand not only how software components are structured and interconnected, but also how these structural patterns evolve over time to influence fault probability.

3.2.4 | Hyperparameter Configuration

3.2.4.1 | Hyperparameter Selection Process. All hyperparameters were selected through a combination of literature review, preliminary experiments, and established best practices for SFP tasks. The selection process prioritised both computational efficiency and model performance across the diverse dataset characteristics in our study.

3.2.4.2 | Optimiser and Learning Rate Configuration. The Adam optimiser [62] was employed across all experiments with a learning rate of 0.0005. This learning rate was selected after preliminary testing with rates ranging from 0.001 to 0.0001, where 0.0005 demonstrated the most stable convergence behaviour across different dataset sizes and model architectures. The choice of Adam optimiser is justified by its adaptive learning rate properties and robust performance in handling sparse gradients, which is particularly relevant for software metrics data that often contains features with varying scales and distributions [63]. Adam's momentum-based approach helps navigate the complex loss landscapes characteristic of both DNN and GNN architectures [64].

3.2.4.3 | Batch Size and Training Configuration. The batch size was consistently set to 32 across all experiments, striking a balance between gradient estimation quality and computational efficiency [65]. This choice was validated through preliminary experiments with batch sizes of 16, 32, and 64, where 32 provided the optimal trade-off between training stability and memory requirements across different dataset sizes. Smaller batch sizes (16) resulted in more noisy gradient updates, while larger batch sizes (64) led to memory constraints for the larger GNN+LSTM models and reduced generalisation performance.

Training was conducted for exactly 10 epochs across all model configurations to ensure fair comparison. This fixed epoch count was determined through validation curves on a subset of datasets, where performance typically plateaued between 8 and 12 epochs. The consistent training duration prevents bias towards any particular architecture that might benefit from longer training periods [66].

3.2.4.4 | Regularisation and Model Architecture Parameters. Dropout was applied with a rate of 0.5 throughout all architectures, following established guidelines for neural network regularisation [57]. This rate has been empirically validated across numerous software engineering applications and provides effective overfitting prevention without overly constraining model capacity [67].

For the GNN+LSTM architectures, the number of attention heads in GAT layers was fixed at 4, following recommendations from graph neural network literature that suggests 4–8 heads provide optimal attention diversity without excessive computational overhead [59]. The LSTM component consistently employed 2 layers with dropout of 0.5 between layers, balancing the model's ability to capture temporal dependencies while preventing overfitting.

3.2.4.5 | Loss Function and Class Imbalance Handling. Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss) was employed for all models due to its numerical stability and effectiveness in binary classification tasks [68]. The selection of this loss function over alternatives such as Focal Loss or weighted cross-entropy was based on preliminary experiments showing comparable performance with reduced computational complexity.

Class imbalance was addressed through different strategies for each architecture type. For GNN+LSTM models, class weights were computed using scikit-learn's "balanced" strategy [69], which calculates weights inversely proportional to class frequencies in the input data. This approach was applied as positive weights in the BCEWithLogitsLoss function, effectively increasing the penalty for misclassifying minority class instances [70].

The decision to use class weights for GNN models rather than sampling techniques was based on the observation that GNNs benefit more from preserving the original graph structure, whereas sampling techniques can disrupt node relationships that are crucial for graph-based learning [71].

3.2.4.6 | Cross-Validation and Reproducibility. Stratified 5-fold cross-validation was implemented to ensure robust performance estimation while maintaining computational feasibility across multiple model configurations [72]. The stratification ensures consistent class distribution across folds, which is particularly important given the inherent class imbalance in software fault datasets.

Random seeds were fixed at 42 throughout all experimental components, including data splitting, model initialisation, and sampling procedures, ensuring complete reproducibility of results [73]. This systematic approach to randomness control enables reliable comparison across different experimental conditions and facilitates replication by other researchers.

3.2.4.7 | Data Preprocessing Integration. The hyperparameter selection process was closely integrated with our data preprocessing pipeline. RandomOverSampler was applied with the same random seed (42) to ensure consistent data augmentation across all experimental runs. Although SMOTE was considered as an alternative class balancing technique, preliminary experiments indicated that the simpler RandomOverSampler

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

approach provided comparable performance with reduced computational overhead, leading to its adoption as the primary class balancing method.

3.2.4.8 | Implementation and Computational Environment. All experiments utilised PyTorch [74] and PyTorch Geometric [75] libraries, with automatic GPU acceleration when available and graceful fallback to CPU computation. This implementation strategy ensures accessibility across different computational environments while maximising training efficiency where hardware permits.

3.3 | Evaluation Measures

To evaluate the performance of the SFP models, we employ multiple evaluation measures including Accuracy, Precision, Recall, and F1-score.

Precision measures the proportion of correctly identified faulty modules among all modules predicted as faulty, while Recall (also known as sensitivity or true positive rate) measures the proportion of actual faulty modules that were correctly identified. Accuracy represents the overall proportion of correctly classified instances across both faulty and non-faulty modules.

The F1-score provides a balanced measure between precision and recall by calculating their harmonic mean. This metric is particularly valuable in our study for several reasons:

- It provides a single score that balances the trade-off between precision and recall.
- It is especially useful in the context of imbalanced datasets, which is common in SFP where faulty modules are typically the minority class.
- It helps in comparing different models more comprehensively than using either precision or recall alone.

3.4 | Data Preprocessing

Data preprocessing is a crucial step in ensuring the reliability and performance of SFP models. Each step in the preprocessing pipeline is carefully designed to address the challenges of software metrics data, including dimensionality, redundancy, class imbalance, and scaling inconsistencies. The preprocessing workflow was implemented using the `scikit-learn` [76] and `imbalanced-learn` [77] libraries.

3.4.1 | Controlled Variations

High-dimensional datasets often pose challenges for predictive modelling, including overfitting and reduced generalisation capabilities. To mitigate these issues in our software metrics data, we implemented a controlled variation approach to address features with constant values.

Our approach first involved identifying columns with constant values in the dataset. For each identified constant column, we introduced controlled variability by randomly selecting and modifying 10% of its instances. To ensure the reproducibility of our experiments, we implemented this random selection using a fixed random seed (`np.random.seed (42)`) from the `numpy` package [78].

This process aligns with theoretical recommendations for mitigating the "curse of dimensionality" by introducing minor but controlled variations in static feature values [55]. The controlled nature of these modifications helps reduce the risk of overfitting while improving model generalisation, particularly important in high-dimensional settings such as SFP where multiple metrics are combined.

3.4.2 | Feature Engineering

To uncover relationships between error-type metrics, we generated interaction features through pairwise multiplication of features with the `ESM_` prefix. For example, from the original metrics `ESM_Arithmetic` and `ESM_NullPointer`, we created a new feature `ESM_Arithmetic_x_ESM_NullPointer`. This approach aligns with recommendations in [55], which emphasise the importance of including interaction terms for capturing relationships between variables in predictive models.

The creation of these interaction terms aims to help the model identify potential dependencies between different types of runtime errors. By explicitly representing these interactions as features, we provide additional information that may be relevant for fault prediction. This feature engineering approach is particularly relevant when working with error-type metrics, as software faults might be associated with combinations of different error patterns rather than individual error types alone.

3.4.3 | Feature Scaling

Feature scaling is an essential step in our preprocessing pipeline, particularly when working with diverse software metrics. All non-constant numerical columns underwent scaling using `scikit-learn`'s `PowerTransformer` with the Yeo-Johnson transformation [79]. This transformation was chosen for its ability to standardise feature distributions to approximate Gaussian distributions, ensuring that no single feature disproportionately influences the model's learning process [55, 80].

The selection of Yeo-Johnson transformation is particularly relevant for our study as we combine conventional software metrics with error-type metrics, which may operate on different scales. The transformation helps normalise these features while preserving their relative relationships. This standardisation is important when working with software metrics data, which often exhibits varying distributions across different types of measurements.

The scaling process helps improve model convergence by reducing feature scale inconsistencies, ensuring that all features can contribute appropriately to the model's learning process regardless of their original scale.

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

3.4.4 | Class Balancing

Our datasets present varying levels of class imbalance that need to be addressed for effective fault prediction. The ANTLR v4 dataset exhibits the most pronounced imbalance, with 80.77% of instances being non-faulty and only 19.23% being faulty, indicating a substantial under-representation of faulty modules. The JUnit4 dataset shows a similar but less severe pattern, comprising 74.29% non-faulty and 25.71% faulty instances. In contrast, both the Elastic Search and OrientDB datasets present more balanced distributions, with Elastic Search containing 55.45% non-faulty and 44.55% faulty instances, while OrientDB shows a similar pattern with 54.60% non-faulty and 45.40% faulty instances.

These varying levels of class imbalance pose significant challenges for model training and evaluation. In particular, imbalanced datasets can lead to biased models that favour the majority class, potentially resulting in poor fault prediction performance for the minority class. This is especially concerning in SFP, where accurately identifying faulty modules is crucial despite their typically lower representation in the dataset.

Out of various resampling techniques, we selected the synthetic minority over-sampling technique (SMOTE) for its optimal balance of simplicity and effectiveness. While more sophisticated algorithms exist, such as Borderline-SMOTE [81], ADASYN [82], and SMOTEENN [83], these approaches introduce additional complexity and computational overhead [84] without guaranteed performance benefits. SMOTE generates synthetic samples by interpolating between existing minority class examples, effectively mitigating overfitting without discarding important data [85]. This approach has demonstrated particular effectiveness in SFP contexts, with studies showing improved classification performance through effective handling of class imbalance [86–88].

In our implementation, SMOTE was applied exclusively to the training data, maintaining the original distribution in the test set. This approach ensures that our evaluation metrics reflect the model's performance on real-world data distributions while allowing the model to learn from a balanced training set. The effectiveness of this strategy is particularly evident in highly imbalanced datasets such as ANTLR v4 and JUnit4, where balanced training data is crucial for developing models capable of identifying the under-represented faulty modules. The choice of SMOTE over alternative techniques is further justified by its ability to enhance model accuracy without introducing the significant drawbacks associated with other methods, making it particularly suitable for our comparative study of DNN and GNN+LSTM architectures.

4 | Experimental Results and Discussions

This section presents our experimental results with comprehensive statistical analysis and critical evaluation addressing the research questions outlined in Section 1. We employ rigorous statistical methods to validate our findings and provide critical insights into the observed patterns, their implications, and inherent limitations. Throughout this section, we employ the following model configuration notation to facilitate clear presentation and comparison:

- **DNN-C:** Deep neural network trained on conventional software metrics dataset.
- **DNN-F:** Deep neural network trained on full dataset (conventional software metrics + error-type metrics).
- **GNN-C:** Graph neural network + LSTM trained on conventional software metrics dataset.
- **GNN-F:** Graph neural network + LSTM trained on full dataset (conventional software metrics + error-type metrics).

These four configurations enable systematic comparison of both architectural choices (DNN vs. GNN+LSTM) and metric set effectiveness (conventional software metrics vs. full metric set including error-type metrics).

4.1 | Architectural Comparison: Statistical Evidence and Critical Analysis

To address RQ1 regarding the comparative performance between DNN and GNN+LSTM architectures, we conducted comprehensive statistical analysis to test and determine whether the observed performance differences were meaningful rather than due to random variation. The Friedman test [89] was selected as our primary statistical method due to its non-parametric nature, which makes no assumptions about underlying data distributions—a critical consideration given the diverse characteristics of our software fault datasets. This test is particularly appropriate for our experimental design where multiple algorithms are evaluated across multiple datasets, effectively handling the dependent nature of our comparisons where all models were assessed on identical data [90].

Our analysis revealed statistically significant differences between the four model configurations for both accuracy ($\chi^2 = 10.200$, $df = 3$, $p < 0.05$) and F1-score ($\chi^2 = 12.000$, $df = 3$, $p < 0.05$). These results provide empirical evidence that the observed performance differences between DNN and GNN+LSTM architectures represent genuine improvements rather than random fluctuations attributable to dataset variance or experimental noise. However, it is crucial to note that statistical significance alone does not necessarily imply practical significance or real-world applicability [91]. The magnitude and consistency of these differences across different software projects require careful examination to determine their practical value for software quality assurance practitioners.

Table 6 presents the mean ranks from the Friedman test, demonstrating a clear hierarchical pattern in model performance. The consistent positioning of GNN-based models in the top two ranks for both metrics provides strong initial evidence for architectural superiority. However, this ranking alone requires deeper interpretation. The relatively close ranks between GNN-C (1.25) and GNN-F (1.75) for accuracy suggest that architectural choice may exert greater influence on performance than metric selection—a finding that challenges conventional assumptions about the primacy of feature engineering in predictive modelling [92].

More concerning is the observation that DNN models consistently occupy the bottom two positions regardless of metric set

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 6 | Model performance rankings from Friedman test.

Model	Accuracy rank	F1-score rank
GNN-C	1.25 (1st)	2.00 (2nd)
GNN-F	1.75 (2nd)	1.00 (1st)
DNN-F	3.25 (3rd)	3.00 (3rd)
DNN-C	3.75 (4th)	4.00 (4th)

Note: Lower ranks indicate superior performance.

employed. This pattern suggests fundamental architectural limitations that cannot be easily overcome through feature engineering alone, supporting recent critiques of traditional deep learning approaches in software engineering tasks [93]. The rankings reveal that GNN+LSTM models maintain their superiority even when trained on conventional metrics alone (GNN-C), indicating that the architectural advantages stem from the models' intrinsic capability to capture software system complexities rather than merely benefiting from enhanced input representations.

Following the significant Friedman test results, we conducted Nemenyi post-hoc tests ($\alpha = 0.05$) to identify specific pairwise differences between model configurations [94]. The Nemenyi test is specifically designed for multiple comparisons following a significant Friedman test, providing robust pairwise comparisons whilst controlling for the increased risk of Type I errors—false positive discoveries—when examining all possible model pairs [90]. With a critical difference (CD) of 2.345 for our four-model comparison, the analysis revealed statistically significant performance differences in two critical comparisons: GNN-C demonstrated significantly superior accuracy compared to DNN-C with a rank difference of 2.500 ($p < 0.05$), whilst GNN-F achieved significantly higher F1-scores than DNN-C with a rank difference of 3.000 ($p < 0.05$).

These pairwise comparisons provide nuanced insights into architectural advantages. The significant difference between GNN-C and DNN-C (rank difference: 2.500) is particularly revealing, as both models were trained on identical conventional metrics, isolating architectural effects from feature engineering considerations. This finding provides empirical support for the theoretical advantages of graph-based representations in capturing software dependencies [95], whilst simultaneously challenging the sufficiency of standard neural network architectures for software engineering tasks.

However, the absence of significant differences in several pairwise comparisons warrants critical examination. The non-significant difference between GNN-C and GNN-F suggests that the sophisticated GNN+LSTM architecture may be reaching a performance ceiling that additional features cannot easily exceed—a phenomenon consistent with recent observations about the diminishing returns of feature augmentation in highly capable architectures [96]. This finding has important implications for resource allocation in software quality assurance: if architectural improvements provide greater benefits than metric engineering, organisations might achieve better results by investing in model sophistication rather than extensive metric collection infrastructures.

TABLE 7 | Effect size analysis using Cohen's d.

Comparison	Mean diff.	Cohen's d	Effect size
Accuracy			
GNN-C vs. DNN-C	+3.9%	1.507	Large
GNN-F vs. DNN-F	+3.3%	1.284	Large
F1-Score			
GNN-C vs. DNN-C	+3.3%	1.395	Large
GNN-F vs. DNN-F	+2.6%	1.091	Large

Note: Cohen's $d > 0.8$ indicates large effect size.

To assess the practical significance of observed performance differences beyond mere statistical detectability, we calculated Cohen's d effect sizes for all pairwise comparisons between model architectures [97, 98]. Cohen's d provides a standardised measure of effect magnitude that enables interpretation of whether statistically significant differences translate into meaningful practical improvements for software quality assurance practitioners [99]. As presented in Table 7, our effect size analysis reveals that all architectural comparisons demonstrate large effect sizes ($d > 0.8$), indicating that the performance differences between GNN+LSTM and DNN models represent substantial practical improvements rather than merely statistically detectable marginal gains.

The comparison between GNN-C and DNN-C yields the largest effect size for accuracy ($d = 1.507$), providing strong evidence that architectural advantages constitute practically significant improvements in SFP performance. According to conventional interpretations of Cohen's d, effect sizes exceeding 1.2 are considered very large and represent differences that would be readily apparent in practical applications [100]. This magnitude suggests that the architectural transition from DNNs to GNN+LSTM models represents a fundamental improvement in fault prediction capability rather than incremental optimisation.

However, a critical examination reveals two important limitations in interpreting these effect sizes. First, the decrease in effect size from GNN-C versus DNN-C ($d = 1.507$) to GNN-F versus DNN-F ($d = 1.284$) for accuracy suggests that metric augmentation may actually attenuate architectural advantages—a counter-intuitive finding discussed in detail in Section 4.2. Second, whilst large effect sizes demonstrate practical significance within our experimental framework, their generalisability to industrial contexts with different characteristics (e.g., different programming languages, development methodologies, or organisational cultures) requires cautious interpretation. The projects in our study, whilst diverse, represent open-source Java systems with specific characteristics that may not fully capture the breadth of software engineering practices in industry.

The consistent finding of large effect sizes across multiple metrics and comparisons provides robust evidence for GNN+LSTM architectural superiority. These findings align with recent theoretical developments in applying graph neural networks to software engineering tasks, where the ability to explicitly model structural dependencies through graph attention mechanisms has been shown to provide fundamental advantages over

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

traditional approaches [95, 101]. The superior performance can be attributed to three key architectural capabilities: (1) graph attention mechanisms that differentially weight the importance of various structural dependencies, enabling the model to focus on fault-relevant relationships; (2) LSTM components that capture temporal evolution patterns in software metrics, modelling how faults emerge through development processes; and (3) the synergistic combination of structural and temporal modelling that provides a more complete representation of software system characteristics than either approach alone.

Nevertheless, these architectural advantages come with important practical trade-offs that practitioners must consider. GNN+LSTM models require substantially greater computational resources for both training and inference, introduce additional hyperparameter complexity, and demand more sophisticated implementation expertise compared to traditional DNNs [71]. The 3%–4% absolute improvement in accuracy and F1-score, whilst statistically and practically significant, must be weighed against these implementation costs. For organisations with severely constrained computational resources or limited machine learning expertise, the simpler DNN architectures might represent a more pragmatic choice despite their inferior performance.

The findings also raise important questions about the theoretical limits of current approaches. The relatively modest absolute improvements (3%–4%) despite large effect sizes suggest that both architectural families may be approaching fundamental performance ceilings imposed by current metric representations and data quality limitations in software fault datasets [2]. Future research must address whether the observed performance plateau reflects inherent limitations in current software metrics or whether more sophisticated architectural innovations might unlock further improvements.

4.2 | Error-Type Metrics Integration: The Multi-Collinearity Challenge

Addressing RQ2, our analysis of error-type metrics integration reveals unexpected and theoretically important findings that challenge conventional assumptions about metric combination strategies in software fault prediction. Table 8 presents the quantitative impact of augmenting conventional software metrics with error-type metrics across different architectural configurations.

The analysis reveals that error-type metrics provide modest benefits for DNN models (mean improvement: +0.7%) but show negligible or slightly negative effects for GNN+LSTM models (mean change: −0.1%). These findings contrast sharply with our previous study [12], where error-type metrics alone demonstrated substantial improvements of up to 40% in AUC and ROC scores compared to conventional metrics. This apparent paradox—where individually effective metrics become ineffective or counterproductive when combined—provides compelling evidence for multi-collinearity issues when integrating different metric families.

TABLE 8 | Impact of error-type metrics integration across architectures.

Dataset	DNN effect	GNN+LSTM effect
ANTLR v4	+2.4%[a]	+0.4%
Elasticsearch	+0.2%	-0.3%
JUnit4	-0.3%	0.0%
OrientDB	+0.4%	-0.6%
Mean	+0.7%	-0.1%

[a]Statistically significant at $p < 0.05$.

Multi-collinearity, defined as linear relationships between predictor variables, represents a fundamental challenge in predictive modelling that can manifest in multiple detrimental ways [102]. When predictor variables capture overlapping aspects of the underlying phenomenon through different measurement approaches, they introduce redundant information that can degrade model performance through several mechanisms: (1) increased variance in coefficient estimates, leading to model instability; (2) difficulty in isolating individual predictor effects, reducing interpretability; (3) potential overfitting as models attempt to learn from correlated noise; and (4) computational inefficiency as models process redundant information [103].

In our context, the multi-collinearity likely arises because error-type metrics and conventional software metrics, whilst measuring different aspects of code quality (runtime error propensity versus structural complexity), capture overlapping fault-proneness signals through correlated indicators. For instance, modules with high cyclomatic complexity (a conventional metric) tend to exhibit elevated error-type counts, creating implicit correlations that introduce redundancy when both metric families are combined [104]. Recent research has demonstrated that deep neural networks, whilst theoretically capable of handling correlated features, often struggle to effectively disentangle highly homogeneous or correlated inputs in practice [105]. This finding aligns with our observation that combined metrics provide minimal additional value over individual metric sets.

The differential response across architectures provides valuable mechanistic insights into how different models handle feature interactions and redundancy. DNN models, with their relatively straightforward feed-forward processing architecture, appear to extract modest utility from additional information sources even when substantial redundancy exists. The observed +0.7% mean improvement, whilst marginal, suggests that simpler architectures can still benefit from feature augmentation despite introducing some multi-collinearity. This behaviour likely reflects DNNs' capacity to learn non-linear combinations of features that partially compensate for redundant information, albeit with reduced efficiency compared to using well-curated individual metric sets.

Conversely, GNN+LSTM models show minimal or negative responses to error-type metrics integration (mean change: −0.1%), revealing a critical limitation in our feature engineering approach. This counterintuitive finding can be explained through the lens of recent theoretical work on graph neural networks in software engineering [96]. The sophisticated feature processing capabilities of GNN+LSTM architectures—specifically, their

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

graph attention mechanisms that weight the importance of different relationships and their LSTM components that model temporal dependencies—enable these models to already extract much of the information that error-type metrics explicitly provide. When error-type metrics are added, the resulting feature space contains substantial redundancy that may actually impede learning by introducing noise and increasing model complexity without corresponding information gain.

This interpretation is supported by recent findings in multicollinearity research demonstrating that advanced machine learning models, including deep neural networks, can experience performance degradation when confronted with highly correlated features, particularly in high-dimensional settings [105]. The degradation mechanism operates through multiple pathways: correlated features can lead models to assign arbitrary and unstable importance weights, increase the risk of overfitting to spurious correlations, and complicate the optimisation landscape by introducing multiple near-equivalent solutions [102].

The implications of these findings extend well beyond our specific study to fundamental questions about feature engineering strategies in software fault prediction. The observation that individually effective metric sets become counterproductive when naively combined challenges the common assumption that "more features are better" in predictive modelling. This finding emphasises the critical importance of systematic feature selection and dimensionality reduction techniques, particularly when combining metrics from multiple measurement frameworks [92]. Recent advances in software metrics research have begun advocating for more parsimonious metric sets that balance information content with minimal redundancy [106], a perspective strongly supported by our empirical results.

Furthermore, our results suggest that the effectiveness of metric combination strategies depends heavily on the underlying model architecture's capability to handle feature interactions. For sophisticated models like GNN+LSTM that already possess strong feature interaction modelling capabilities, metric augmentation may introduce diminishing or negative returns. This finding has important practical implications for software quality assurance: organisations employing advanced predictive models might achieve better results by focusing on high-quality individual metric sets rather than attempting to combine multiple measurement frameworks. The computational and organisational costs of collecting, maintaining, and processing multiple metric families may not be justified if the resulting prediction improvements are negligible or negative.

These observations also highlight a critical methodological limitation in software fault prediction research: the tendency to evaluate feature sets in isolation without systematic investigation of interaction effects when combined. Future research must move beyond univariate feature evaluation towards more sophisticated approaches that explicitly account for feature correlations and redundancy. Techniques such as Variance Inflation Factor (VIF) analysis [107], principal component analysis for dimensionality reduction [108], or more advanced methods like mutual information-based feature selection [109] should become standard practice before combining metric sets from different measurement frameworks.

4.3 | Class Imbalance Mitigation: Dataset-Dependent Effectiveness

Our analysis of class balancing effectiveness addresses RQ3 by examining how different model architectures respond to the Synthetic Minority Oversampling Technique (SMOTE) for addressing class imbalance in software fault datasets. The statistical analysis demonstrates significant and consistent improvements across most model configurations, with effectiveness varying substantially based on dataset characteristics and model sophistication.

Table 9 demonstrates that three of the four model configurations achieve statistically significant improvements from class balancing, with the DNN-C model approaching significance ($p = 0.12$). The consistent benefits across different architectures validate the critical importance of addressing class imbalance in SFP. However, these aggregate statistics mask important nuances regarding when and for which models class balancing provides the greatest benefit.

The differential magnitude of improvements across architectures reveals important insights into model robustness to class imbalance. DNN models show larger mean improvements (DNN-C: +7.1%, DNN-F: +5.7%) compared to GNN+LSTM models (GNN-C: +3.9%, GNN-F: +4.0%), suggesting that the sophisticated GNN+LSTM architecture provides some inherent robustness to class imbalance even without explicit balancing techniques. This finding aligns with recent theoretical work demonstrating that graph neural networks' attention mechanisms can naturally adapt to class imbalance by learning to focus on minority class patterns [110]. However, the continued statistical significance of improvements even for GNN+LSTM models demonstrates that class imbalance remains a fundamental challenge requiring explicit attention regardless of architectural sophistication.

The near-significance result for DNN-C ($p = 0.12$) warrants critical examination. Whilst this p-value exceeds the conventional $\alpha = 0.05$ threshold, it represents borderline evidence that should not be dismissed entirely [111]. The mean improvement of +7.1% for DNN-C, higher than any other configuration, suggests practical benefit despite marginal statistical uncertainty. This pattern may reflect higher variance in DNN-C performance across datasets rather than absence of true effect, highlighting the importance of considering both statistical significance and practical effect magnitude in empirical software engineering research [91].

TABLE 9 | Statistical analysis of class balancing effects.

Model	Mean improvement	t-statistic	Significance
DNN-C	+7.1%	1.994	ns
DNN-F	+5.7%	2.843	*
GNN-C	+3.9%	2.556	*
GNN-F	+4.0%	2.738	*

Abbreviation: ns, not significant ($p = 0.12$).
*$p < 0.05$.

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 10 | Class balancing effects stratified by dataset imbalance severity.

Dataset	Mean improvement	Cohen's *d*	Effect size
ANTLR v4	+11.8%	1.721	Large
JUnit4	+15.3%	1.530	Large
OrientDB	+3.1%	0.789	Medium
Elasticsearch	+1.8%	0.650	Medium

Note: Large: $d > 0.8$, Medium: $0.5 < d \leq 0.8$.

The dataset-stratified analysis in Table 10 reveals a clear and theoretically grounded relationship between class imbalance severity and balancing effectiveness. Datasets with more severe class imbalance (ANTLR v4: 80.77% non-faulty, JUnit4: 74.29% non-faulty) demonstrate large effect sizes ($d > 1.5$), whilst more balanced datasets (OrientDB: 54.60% non-faulty, Elasticsearch: 55.45% non-faulty) show medium effects ($0.5 < d < 0.8$). This monotonic relationship validates theoretical expectations that sampling techniques provide greatest benefit when addressing substantial class distribution skew [84].

However, this clear pattern also highlights important limitations in SMOTE's applicability and effectiveness. The substantial differences in effect sizes across datasets (ranging from $d = 0.650$ for Elasticsearch to $d = 1.721$ for ANTLR v4) suggest that the optimal class balancing strategy may be dataset-dependent. Recent research in class imbalance learning has demonstrated that SMOTE's effectiveness varies based on multiple dataset characteristics beyond simple imbalance ratio, including class overlap, dimensionality, and sample density in feature space [112]. Our results support these findings, suggesting that practitioners should carefully assess their specific dataset characteristics before selecting class balancing approaches.

The large improvements observed for severely imbalanced datasets (ANTLR v4: +11.8%, JUnit4: +15.3%) must be interpreted with appropriate caution regarding their practical implications. Whilst these improvements are both statistically significant and practically substantial, they come with important trade-offs. SMOTE, by generating synthetic minority class samples through interpolation, can potentially introduce several problems: (1) overfitting to interpolated examples that may not represent real-world fault patterns; (2) blurring of class boundaries when minority and majority class regions overlap in feature space; (3) amplification of label noise if the original minority class samples contain mislabelled instances [83]. These concerns are particularly relevant in software fault prediction, where the minority class (faulty modules) may exhibit substantial within-class heterogeneity due to diverse fault mechanisms.

Furthermore, our use of SMOTE exclusively on training data, whilst methodologically sound for preventing optimistic bias in evaluation metrics, raises questions about the ecological validity of our findings. In practical software quality assurance scenarios, prediction models must operate on naturally occurring class distributions. The substantial performance improvements observed on SMOTE-augmented training data may not translate proportionally to real-world deployment scenarios where severe class imbalance persists [113]. This gap between training-time interventions and deployment-time realities represents a critical limitation that warrants careful consideration by practitioners.

The differential effectiveness across model architectures also provides mechanistic insights into how different models respond to synthetic sample generation. The larger improvements observed for simpler DNN models compared to sophisticated GNN+LSTM architectures suggest that more complex models may be more resilient to class imbalance through their enhanced representational capacity. However, this resilience comes with the caveat that even sophisticated models show statistically significant improvements from SMOTE, indicating that class imbalance remains a fundamental challenge that cannot be fully addressed through architectural sophistication alone.

Recent research has identified important limitations in SMOTE's effectiveness for high-dimensional data, where the curse of dimensionality can make Euclidean distance-based sample generation less reliable [114]. Our software metrics datasets, whilst not extremely high-dimensional by modern machine learning standards, contain sufficient dimensions (25–30 features in the Full Dataset) that this consideration becomes relevant. The fact that SMOTE demonstrates effectiveness despite these theoretical concerns suggests either that our feature spaces maintain reasonable structure at observed dimensionalities or that the benefits of addressing class imbalance outweigh the costs of imperfect synthetic sample generation.

The practical implications of these findings for software quality assurance are multifaceted. For severely imbalanced datasets (imbalance ratio > 3:1), our results provide strong evidence supporting the use of SMOTE or similar oversampling techniques. The large effect sizes ($d > 1.5$) and substantial absolute improvements (10%–15%) justify the additional computational costs and potential risks of synthetic sample generation. However, for moderately imbalanced datasets (imbalance ratio < 2:1), the benefits become less clear-cut. The medium effect sizes ($d \approx 0.65 - 0.79$) and modest absolute improvements (2%–3%) may not justify the added complexity, particularly for organisations with limited machine learning expertise or computational resources.

These findings also highlight important methodological considerations for future software fault prediction research. The substantial variability in class balancing effectiveness across datasets (effect sizes ranging from 0.650 to 1.721) underscores the importance of reporting results stratified by dataset characteristics rather than only presenting aggregate statistics. Future studies should systematically investigate how dataset properties—including imbalance ratio, class overlap, dimensionality, and sample size—moderate the effectiveness of different class balancing approaches. This more nuanced understanding would enable evidence-based guidance for practitioners selecting appropriate techniques for their specific contexts.

4.4 | Implications for SFP Practice

Our comprehensive statistical analysis yields several important methodological insights that extend beyond the specific architectural and metric comparisons to broader questions about

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

conducting rigorous empirical software engineering research. The consistent application of non-parametric statistical tests (Friedman test), post-hoc pairwise comparisons (Nemenyi test), and effect size calculations (Cohen's d) represents a methodological template for robust evaluation of machine learning approaches in software engineering contexts [90, 115].

The large effect sizes observed for architectural comparisons ($d > 1.0$) provide strong evidence that the GNN+LSTM architecture offers practically significant improvements over traditional DNNs. However, the absolute magnitude of these improvements (3%–4% in accuracy and F1-score) raises important questions about whether current approaches are approaching fundamental performance ceilings. This concern is amplified by comparing our current results with our previous study [12], where error-type metrics alone achieved superior performance compared to the combined metric approach examined here—a finding discussed in detail in Section 4.5.

The unexpected finding that metric combination can degrade performance compared to individual metric sets has profound implications for feature engineering practices in software fault prediction. This observation challenges the prevalent assumption that additional features inevitably improve model performance, instead suggesting that naive feature augmentation may introduce multi-collinearity and noise that sophisticated architectures cannot fully overcome [92]. Future research must move towards more principled approaches to feature combination that explicitly account for potential redundancy and correlation between different metric families.

The differential response of different architectures to error-type metrics integration provides mechanistic insights into how model sophistication affects feature engineering requirements. Simpler DNN models show modest benefits from additional features (+0.7% mean improvement), whilst sophisticated GNN+LSTM models show negligible or negative effects (−0.1% mean change). This pattern suggests that feature engineering recommendations should be architecture-dependent: simpler models may benefit from feature augmentation despite some redundancy, whilst sophisticated models require more careful feature curation to avoid introducing counterproductive noise.

The class imbalance mitigation results demonstrate that SMOTE remains an effective technique for addressing severe class imbalance, with improvements ranging from 1.8% to 15.3% depending on dataset characteristics. However, the substantial variability in effectiveness across datasets (effect sizes: $d = 0.650$ to $d = 1.721$) highlights the importance of dataset-specific evaluation rather than blanket application of class balancing techniques. Practitioners should assess their specific class distribution characteristics and conduct pilot studies to determine whether the benefits of synthetic sample generation justify the implementation complexity and potential risks of overfitting to interpolated examples.

Our statistical framework, combining Friedman tests for overall comparisons, Nemenyi post-hoc tests for pairwise analysis, and Cohen's d for effect size quantification, addresses growing concerns about reproducibility and statistical validity in empirical software engineering research [116]. This comprehensive approach enables robust conclusions about both statistical significance (whether observed differences are unlikely due to chance) and practical significance (whether observed differences matter for real-world applications). Future empirical studies in software engineering should adopt similar comprehensive statistical frameworks rather than relying solely on point estimates of performance metrics.

4.5 | Comparison With Previous Study and Limitations

To provide comprehensive context for our findings, we critically compare our results with our previous study [12], where DNN models were trained separately on CK Metrics and error-type metrics. This comparison, supported by our statistical analysis, reveals important limitations in our current approach and raises fundamental questions about optimal feature engineering strategies in software fault prediction.

Figures 2 and 3 present detailed performance comparisons across all model configurations and datasets, providing visual evidence for patterns identified through statistical testing. Figure 4 illustrates precision-recall trade-offs, demonstrating how different architectures balance these critical performance aspects.

Our statistical analysis provides evidence for concerning patterns when comparing current results with previous work. Despite employing more sophisticated GNN+LSTM architectures that demonstrate statistically significant improvements over baseline DNN approaches within the current study (confirmed through Friedman tests: $\chi^2 = 12.000$ for F1-score, $p < 0.05$), the absolute performance levels achieved are lower than those obtained in our previous study using simpler architectures with individual metric sets.

For the ANTLR v4 dataset, our previous study achieved superior results with error-type metrics alone: F1-score of 0.881 and accuracy of 0.789, compared to CK Metrics' F1-score of 0.835 and accuracy of 0.741. In contrast, both current DNN and GNN+LSTM models trained on combined metrics show degraded absolute performance. The DNN-F model achieves F1-score of 0.753 and accuracy of 0.734, whilst the GNN+LSTM-F model performs marginally better with F1-score of 0.781 and accuracy of 0.759—yet both fall short of the previous study's error-type metrics results.

This pattern is statistically consistent across datasets, providing strong evidence for multi-collinearity issues in combined metric sets discussed in Section 4.2. The JUnit dataset exhibits similar performance degradation: whilst our previous study achieved F1-scores of 0.837 and 0.836 with error-type metrics and CK Metrics respectively (with accuracies of 0.741 and 0.722), the current combined metrics approach yields substantially lower performance. The DNN-F model achieves F1-score of 0.679 and accuracy of 0.704, whilst the GNN+LSTM-F model achieves F1-score of 0.729 and accuracy of 0.737—both falling short of individual metric set performance.

The OrientDB dataset presents the only exception to this pattern, where the GNN+LSTM-F model shows marginal improvement with F1-score of 0.754 and accuracy of 0.732, compared

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

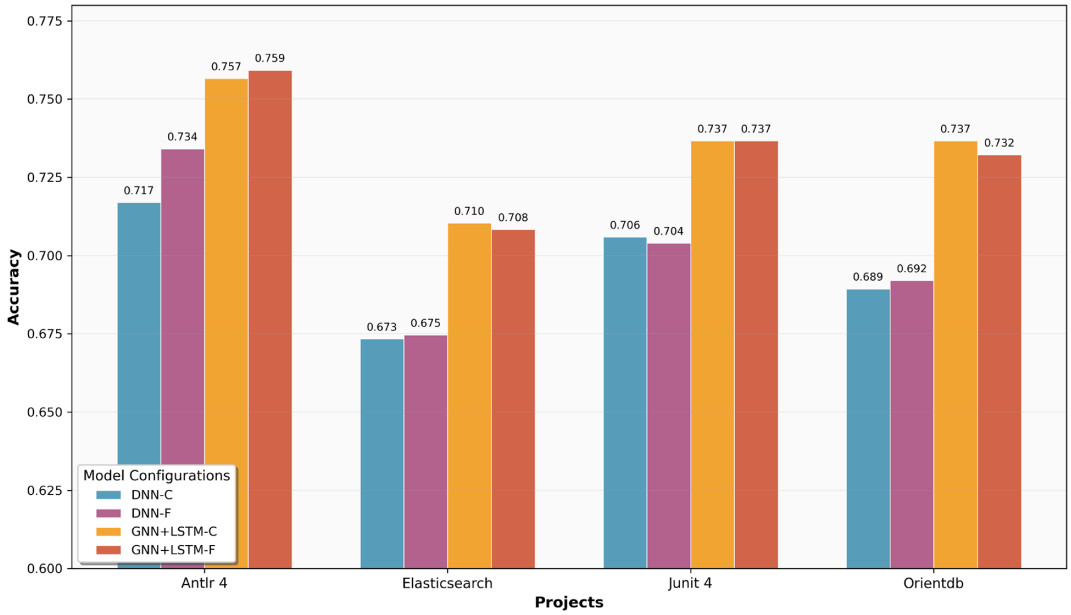


FIGURE 2 | Comparison of model accuracy across datasets.

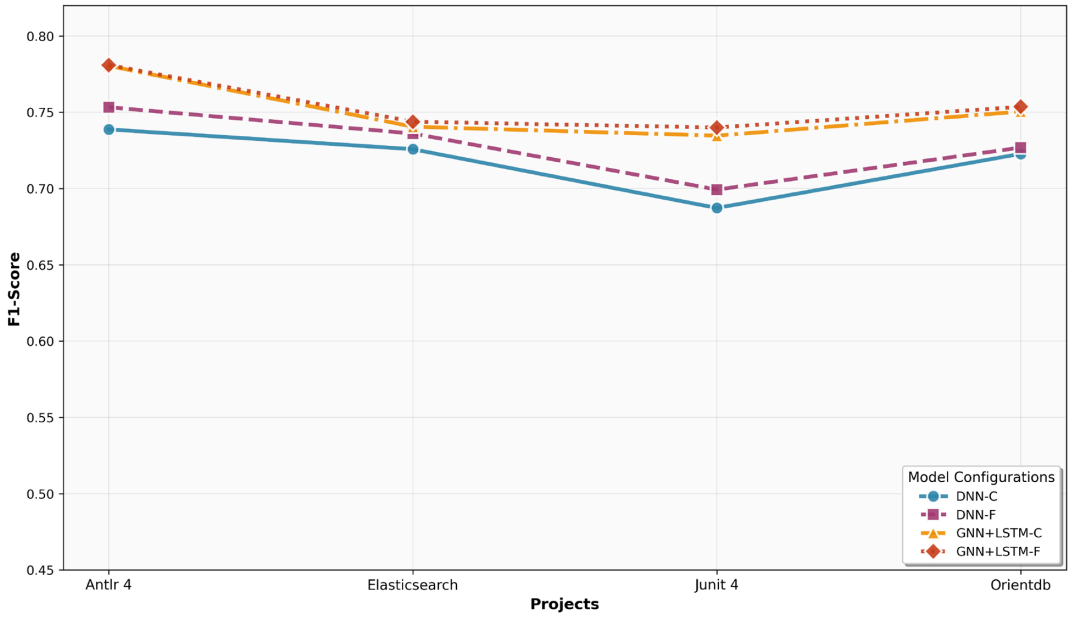


FIGURE 3 | F1-score comparison across models and datasets.

to previous study's F1-scores of 0.727 with error-type metrics and 0.708 with CK Metrics. This exception provides valuable insight into conditions under which metric combination might be beneficial. OrientDB's larger dataset size (5514 modules) and more balanced class distribution (54.60% non-faulty) may enable sophisticated GNN+LSTM architectures to more effectively manage feature interactions and extract complementary information from combined metrics. This observation suggests that dataset characteristics—particularly size and class balance—moderate the effectiveness of metric combination strategies.

For the Elasticsearch dataset, performance degradation patterns persist despite substantial dataset size (18,324 modules). Our previous work achieved F1-scores of 0.855 with error-type metrics and 0.810 with CK Metrics, with corresponding accuracies of 0.705 and 0.693. The current study's approaches show reduced performance, with GNN+LSTM-F achieving F1-score of 0.744 and accuracy of 0.708, falling short of error-type metrics alone performance by approximately 11 percentage points in F1-score.

These findings create an apparent paradox that demands critical examination: whilst our statistical tests demonstrate clear and significant advantages of GNN+LSTM architectures over DNN approaches within the current experimental framework (Friedman $\chi^2 = 12.000$ for F1-score, $p < 0.05$; large effect sizes $d > 1.0$ for architectural comparisons), the absolute performance levels achieved are systematically lower than those obtained

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

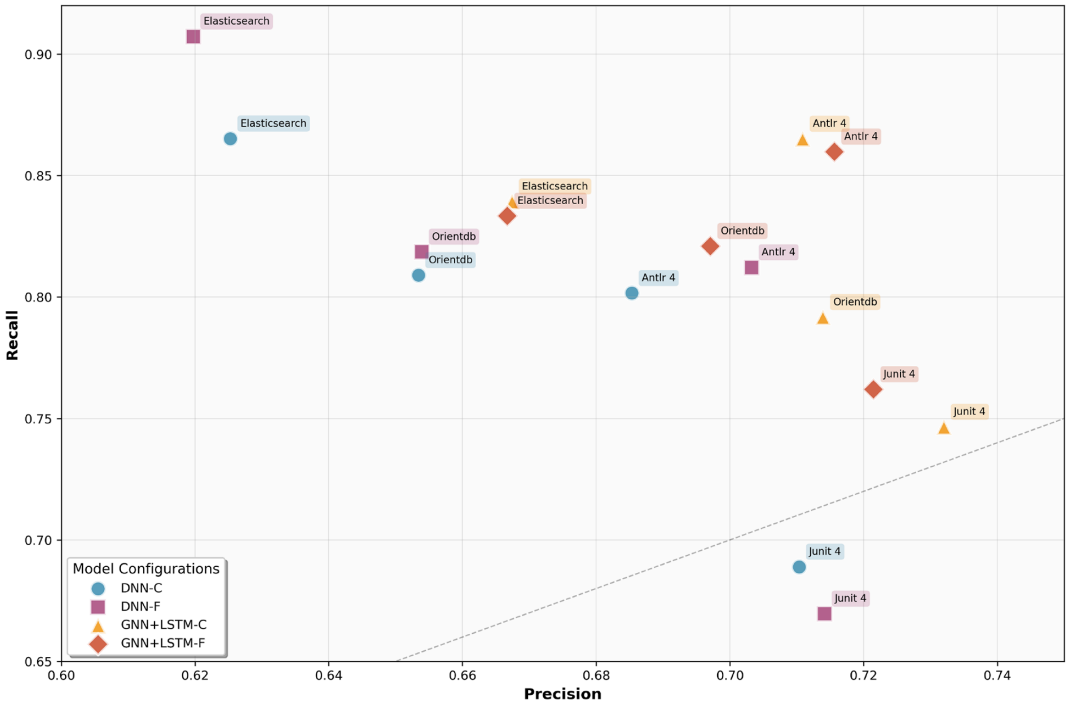


FIGURE 4 | Precision-recall trade-offs across model configurations.

using simpler architectures with individual metric sets in our previous study. This paradox highlights the critical importance of experimental design choices—particularly regarding feature engineering and metric selection—that may exert larger impacts on absolute performance than architectural sophistication alone.

The consistent pattern of performance degradation across projects when combining metrics, despite architectural improvements, provides compelling evidence for multi-collinearity issues in our combined feature set. Multi-collinearity manifests when independent variables exhibit linear relationships, potentially leading to: (1) reduced model stability due to inflated variance in parameter estimates; (2) difficulty isolating individual metric contributions, complicating model interpretation; (3) potential overfitting to redundant information that does not generalise; and (4) computational inefficiency from processing correlated features [102, 103].

Our combined metric space likely introduces such multi-collinearity because error-type metrics and conventional software metrics, whilst measuring conceptually distinct aspects (runtime error propensity versus structural complexity), capture overlapping fault-proneness signals through correlated indicators. Recent research has demonstrated that deep neural networks struggle to effectively disentangle highly correlated or homogeneous input features, leading to suboptimal performance even when individual feature sets prove effective in isolation [105]. This theoretical understanding aligns with our empirical observations and provides mechanistic explanation for the observed performance paradox.

The large effect sizes observed for architectural comparisons within the current study ($d > 1.0$) confirm that models are learning different patterns and that GNN+LSTM architectures provide genuine improvements over traditional DNNs. However, these within-study improvements occur within a constrained solution space defined by our combined metric set. The fact that simpler architectures with better-curated feature sets achieved superior absolute performance suggests that feature engineering may represent a more critical determinant of practical performance than architectural sophistication—at least within the range of architectures examined.

This interpretation challenges prevalent narratives in machine learning research that emphasise architectural innovation as the primary driver of progress. Our results suggest that thoughtful feature curation, explicit consideration of feature correlations, and systematic dimensionality reduction may yield greater practical benefits than adopting increasingly sophisticated model architectures. This finding has important resource allocation implications for software quality assurance organisations: investments in comprehensive metric collection infrastructures and sophisticated model architectures may yield diminishing returns if fundamental issues of feature redundancy and multi-collinearity are not adequately addressed.

The apparent contradiction between within-study statistical superiority and cross-study absolute performance comparison also illustrates important methodological considerations for empirical software engineering research. Statistical tests provide valid evidence for relative performance differences within controlled experimental conditions but may not capture the full complexity of practical deployment scenarios. Our Friedman and Nemenyi tests appropriately identify GNN+LSTM superiority over DNNs within our experimental framework, but comparison with previous work reveals that this framework itself may be suboptimal. This observation emphasises the critical importance of considering both within-study relative comparisons and

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

cross-study absolute benchmarks when evaluating progress in software fault prediction.

These findings highlight several critical limitations and threats to validity in our study. First, the combination of different metric sets introduced complex feature interactions that current model architectures struggle to handle effectively, despite their theoretical capacity for non-linear feature learning. This limitation suggests that our current architectural approaches, whilst sophisticated, may not be optimal for handling the specific challenges posed by combined metric spaces in software fault prediction.

Second, our current architectures, despite demonstrating statistical superiority within this study, may require fundamental modifications to effectively process combined metric sets. The assumption that graph attention mechanisms and LSTM temporal modelling would naturally handle metric interactions proved insufficient. Future architectural innovations might incorporate explicit feature selection mechanisms, regularisation strategies specifically designed for correlated features, or hierarchical attention mechanisms that can differentially process different metric families [117].

Third, our experimental design choices—particularly the decision to combine all metrics without systematic feature selection or dimensionality reduction—may have constrained the solution space in ways that prevented both DNN and GNN+LSTM models from achieving their full potential. This limitation emphasises the critical importance of pilot studies and feature correlation analysis before committing to full-scale experimental evaluation.

These observations suggest several critical directions for future research. Detailed investigation of metric interactions using techniques such as Variance Inflation Factor (VIF) analysis [107], correlation matrices, and mutual information calculations could help identify optimal feature combinations that avoid multi-collinearity issues. Systematic application of dimensionality reduction techniques such as principal component analysis [108], feature selection through mutual information [109], or regularised models that automatically handle correlated features [55] might enable effective metric combination.

Alternative architectural approaches specifically designed for correlated features warrant investigation. Potential candidates include: (1) hierarchical architectures that process different metric families separately before integration; (2) attention mechanisms explicitly designed to identify and downweight redundant features; (3) ensemble approaches that combine predictions from models trained on different metric subsets; and (4) adversarial training approaches that encourage models to learn complementary rather than redundant representations from different metric families.

The methodological implications extend to evaluation practices in software fault prediction research. Our results demonstrate that comparative studies should consider both within-study statistical comparisons and absolute performance benchmarks from previous work. The statistical framework employed in this study—combining Friedman tests for overall comparisons, Nemenyi tests for pairwise analysis, and Cohen's d for effect size quantification—provides valuable evidence for relative performance differences. However, absolute performance comparisons remain essential for assessing practical progress in the field and identifying potential regression in performance due to suboptimal experimental design choices.

These findings also emphasise the critical importance of feature engineering and metric selection as potentially more impactful factors than architectural sophistication in determining practical fault prediction performance. Whilst architectural innovations continue to provide value, the software fault prediction community might achieve greater practical progress through systematic investigation of optimal metric combinations, explicit treatment of feature correlations, and development of theoretically grounded feature selection methodologies. This perspective suggests a research agenda that balances architectural innovation with fundamental feature engineering considerations—a balance that current literature may not adequately maintain.

5 | Conclusion

This study has conducted a comprehensive investigation into the effectiveness of different deep learning architectures for SFP, employing statistical analysis including Friedman tests, Nemenyi post-hoc comparisons, and effect size calculations to validate our findings. Our investigation focused on the comparative analysis of DNN and GNN+LSTM models, the impact of error-type metrics when combined with conventional software metrics, and the effectiveness of class balancing techniques in addressing imbalance issues.

Our statistical analysis provides strong evidence for the superiority of GNN+LSTM architectures over traditional DNN approaches. The Friedman test revealed statistically significant differences between model configurations for both accuracy ($\chi^2 = 10.200,\ p < 0.05$) and F1-score ($\chi^2 = 12.000,\ p < 0.05$). Nemenyi post-hoc tests confirmed specific pairwise superiority of GNN models over DNN models, with effect size analysis demonstrating large practical significance (Cohen's $d > 1.0$) for all architectural comparisons. These findings provide evidence that GNN+LSTM architectures' ability to simultaneously capture structural dependencies through graph attention mechanisms and temporal patterns through LSTM processing leads to meaningful improvements in SFP capability.

The statistical framework also validates our second key finding regarding the unexpected challenges in error-type metrics integration. Despite the architectural improvements demonstrated within our study, comparison with our previous work [12] reveals a concerning pattern where combining error-type metrics with conventional software metrics leads to performance degradation compared to using error-type metrics alone. This finding, supported by our systematic analysis showing minimal benefits from metric combination (mean DNN improvement: +0.7%, mean GNN improvement: −0.1%), highlights the complexity of feature interactions in SFP and provides statistical evidence for multi-collinearity issues when combining different metric sets.

Our statistical analysis of class balancing effectiveness demonstrates significant improvements across most model configurations, with three of four models achieving statistical

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

significance ($p < 0.05$) in paired t-tests. The effect size analysis reveals a clear relationship between class imbalance severity and balancing effectiveness, with severely imbalanced datasets (ANTLR v4 and JUnit4) showing large effect sizes ($d > 1.5$), whilst more balanced datasets demonstrate medium effects ($0.5 < d < 0.8$). These findings provide statistical validation for the critical importance of addressing class imbalance in SFP.

The unexpected performance degradation when combining different metric sets, despite statistically significant architectural improvements within the study, suggests that our current approach to feature integration requires substantial refinement. The apparent paradox between within-study statistical superiority and absolute performance compared to previous work emphasises that experimental design choices, particularly regarding feature selection and combination, may have larger impacts on performance than architectural sophistication alone.

Our findings reveal several critical directions for future research in software fault prediction. The identification of multi-collinearity issues when combining error-type metrics with conventional software metrics necessitates systematic investigation of metric interactions using Variance Inflation Factor analysis, correlation matrices, and mutual information calculations to identify optimal feature combinations that avoid redundancy. Future work should develop architecture-aware feature selection methodologies that explicitly consider model capabilities, given our findings that sophisticated GNN+LSTM models show sensitivity to feature redundancy whilst simpler DNNs benefit from additional features despite some multi-collinearity. Advanced architectural innovations warrant investigation, including hierarchical architectures that process different metric families separately before integration, attention mechanisms explicitly designed to identify and downweight redundant features, ensemble approaches combining predictions from models trained on different metric subsets, and adversarial training approaches encouraging models to learn complementary representations from different metric families.

The substantial variability in class balancing effectiveness and the exceptional performance pattern observed in the OrientDB dataset suggest that dataset characteristics moderate technique effectiveness in ways not yet fully understood. Future research should systematically investigate how dataset properties—including size, class imbalance ratio, class overlap, dimensionality, and domain characteristics—influence the relative effectiveness of different architectural and preprocessing choices. Cross-project and cross-language validation studies are essential to determine whether our findings regarding GNN+LSTM superiority and multi-collinearity challenges generalise to other programming languages, development contexts such as commercial versus open-source environments, and application domains beyond those examined in this study. Additionally, whilst GNN+LSTM models demonstrate superior performance, their substantial computational requirements and implementation complexity warrant investigation of techniques for reducing computational costs whilst maintaining performance benefits, such as model compression, knowledge distillation, or efficient graph neural network variants designed specifically for software engineering applications.

The black-box nature of both DNN and GNN+LSTM models limits their practical adoption in industrial contexts where model decisions must be explainable to stakeholders. Future work should develop interpretability techniques specifically designed for graph-based fault prediction models, enabling practitioners to understand which structural dependencies and temporal patterns drive fault predictions. Beyond static code metrics, future research should investigate incorporating additional information sources such as software evolution history, developer expertise profiles, code review comments, issue tracking data, and continuous integration logs, leveraging the GNN+LSTM architecture's capability to model complex relationships between diverse data sources. Systematic comparison of alternative class balancing approaches including ADASYN, Borderline-SMOTE, ensemble sampling methods, and cost-sensitive learning frameworks would determine optimal strategies for different dataset characteristics and imbalance severities, building upon our findings regarding SMOTE's dataset-dependent effectiveness.

Our study demonstrates that whilst advanced deep learning architectures provide statistically significant and practically meaningful improvements within controlled experimental settings, the broader challenge of SFP requires careful attention to experimental design, feature engineering, and validation methodology. The statistical framework employed in this study—combining Friedman tests for overall comparisons, Nemenyi post-hoc tests for pairwise analysis, and Cohen's d for effect size quantification—provides a template for robust evaluation in empirical software engineering research.

The findings suggest that future advancements in SFP will require integrated approaches that combine architectural innovation with sophisticated feature engineering and methodological improvements. Our statistical analysis provides evidence that architectural choice significantly affects performance, but also demonstrates that feature selection and experimental design decisions may have even larger impacts on practical effectiveness.

The methodological insights from our study extend beyond SFP to broader machine learning applications in software engineering. The multi-collinearity challenges revealed through metric combination, the dataset-dependent effectiveness of preprocessing techniques, and the importance of comprehensive statistical validation all represent critical considerations for empirical research in this domain.

Whilst our study demonstrates the potential of advanced deep learning architectures and validates the importance of addressing class imbalance in SFP, it also reveals the complexity of conducting valid comparative studies and the critical importance of considering multiple evaluation perspectives. The statistical evidence supports the superiority of GNN+LSTM architectures within our experimental framework whilst simultaneously highlighting the need for more sophisticated approaches to feature engineering and experimental design in SFP research.

Disclosure

The authors have nothing to report.

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Endnotes

[1] https://github.com/dangkhoa0303/Error-type-Metrics-Datasets.git.

References

1. G. Destefanis, S. Counsell, G. Concas, and R. Tonelli, "Software Metrics in Agile Software: An Empirical Study," in *Agile Processes in Software Engineering and Extreme Programming: 15th International Conference, XP 2014* (Springer, 2014), 157–170.

2. K. Bhandari, K. Kumar, and A. L. Sangal, "Data Quality Issues in Software Fault Prediction: A Systematic Literature Review," *Artificial Intelligence Review* 56, no. 8 (2023): 7839–7908.

3. L. Qiao, X. Li, Q. Umer, and P. Guo, "Deep Learning Based Software Defect Prediction," *Neurocomputing* 385 (2020): 100–110.

4. L. Šikić, A. S. Kurdija, K. Vladimir, and M. Šilić, "Graph Neural Network for Source Code Defect Prediction," *IEEE Access* 10 (2022): 10402–10415.

5. H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A Semantic Lstm Model for Software Defect Prediction," *IEEE Access* 7 (2019): 83812–83824.

6. Y. M. Ramya, K. Deepthi, A. Vamsai, A. J. Sai, N. Sharma, and B. R. Reddy, "Software Fault Prediction Using Deep Neural Networks," in *Machine Vision and Augmented Intelligence: Select Proceedings of MAI 2022*, vol. 1007 (Springer Nature, 2023), 267.

7. R. Sepahvand, R. Akbari, and S. Hashemi, "Predicting the Bug Fixing Time Using Word Embedding and Deep Long Short Term Memories," *IET Software* 14, no. 3 (2020): 203–212.

8. M. M. Jouybari, A. Tajary, M. Fateh, and V. Abolghasemi, "A Novel Deep Neural Network Structure for Software Fault Prediction," *PeerJ Computer Science* 10 (2024): e2270.

9. H. Ghinaya, R. Herteno, M. R. Faisal, A. Farmadi, and F. Indriani, "Analysis of Important Features in Software Defect Prediction Using Synthetic Minority Oversampling Techniques (SMOTE), Recursive Feature Elimination (RFE) and Random Forest," *Journal of Electronics, Electromedical Engineering, and Medical Informatics* 6, no. 3 (2024): 276–288.

10. T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai, "Software Complexity Analysis Using Halstead Metrics," in *2017 International Conference on Trends in Electronics and Informatics (ICEI)* (IEEE, 2017), 1109–1113.

11. K. Phung, E. Ogunshile, and M. Aydin, "Error-Type a Novel Set of Software Metrics for Software Fault Prediction," *IEEE Access* 11 (2023): 30562–30574.

12. K. Phung, E. Ogunshile, and M. E. Aydin, "Leveraging Deep Learning for Enhanced Software Fault Prediction Using Error-Type Metrics," in *International Symposium on Intelligent and Distributed Computing* (Springer, 2024), 167–179.

13. P. Goodman, *The Practical Implementation of Software Metrics* (McGraw-Hill, Inc, 1993).

14. J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering* 28, no. 1 (2002): 4–17.

15. N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach* (CRC press, 2014).

16. S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering* 20, no. 6 (1994): 476–493.

17. D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using Metrics to Evaluate Software System Maintainability," *Computer* 27, no. 8 (1994): 44–49.

18. F. Rahman and P. Devanbu, "How, and Why, Process Metrics Are Better," in *2013 35th International Conference on Software Engineering (ICSE)* (IEEE, 2013), 432–441.

19. S. S. Rathore and S. Kumar, "A Study on Software Fault Prediction Techniques," *Artificial Intelligence Review* 51 (2019): 255–327.

20. S. Karim, H. L. H. S. Warnars, F. L. Gaol, et al., "Software Metrics for Fault Prediction Using Machine Learning Approaches: A Literature Review With Promise Repository Dataset," in *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, vol. 2017 (IEEE, 2017), 19–23.

21. K. Phung, E. Ogunshile, and M. Aydin, "A Novel Software Fault Prediction Approach to Predict Error-Type Proneness in the Java Programs Using Stream x-Machine and Machine Learning," in *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)* (IEEE, 2021), 168–179.

22. D. Dranidis, K. Bratanis, and F. Ipate, "Jsxm: A Tool for Automated Test Generation," in *International Conference on Software Engineering and Formal Methods* (Springer, 2012), 352–366.

23. P. Kumudha and R. Venkatesan, "Cost-Sensitive Radial Basis Function Neural Network Classifier for Software Defect Prediction," *Scientific World Journal* 2016, no. 1 (2016): 2401496.

24. X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep Learning for Just-In-Time Defect Prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security* (IEEE, 2015), 17–26.

25. C. Manjula and L. Florence, "Deep Neural Network Based Hybrid Approach for Software Defect Prediction Using Software Metrics," *Cluster Computing* 22, no. Suppl 4 (2019): 9847–9863.

26. T. Y. Yu, C. Y. Huang, and N. C. Fang, "Use of Deep Learning Model With Attention Mechanism for Software Fault Prediction," in *2021 8th International Conference on Dependable Systems and Their Applications (DSA)* (IEEE, 2021), 161–171.

27. J. Humphreys and H. K. Dam, "An Explainable Deep Model for Defect Prediction," in *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)* (IEEE, 2019), 49–55.

28. S. Jiang, Y. Chen, Z. He, Y. Shang, and L. Ma, "Cross-Project Defect Prediction via Semantic and Syntactic Encoding," *Empirical Software Engineering* 29, no. 4 (2024): 1–39.

29. S. Albahli, "A Deep Ensemble Learning Method for Effort-Aware Just-In-Time Defect Prediction," *Future Internet* 11, no. 12 (2019): 246.

30. J. Chen, J. Xu, S. Cai, X. Wang, H. Chen, and Z. Li, "Software Defect Prediction Approach Based on a Diversity Ensemble Combined With Neural Network," *IEEE Transactions on Reliability* 73 (2024): 1487–1501.

31. C. Liu, D. Yang, X. Xia, M. Yan, and X. Zhang, "A Two-Phase Transfer Learning Model for Cross-Project Defect Prediction," *Information and Software Technology* 107 (2019): 125–136.

32. O. P. Omondiagbe, S. A. Licorish, and S. G. MacDonell, "Improving Transfer Learning for Software Cross-Project Defect Prediction," *Applied Intelligence* 54, no. 7 (2024): 5593–5616.

33. S. A. Gamel, S. S. Ghoneim, and Y. A. Sultan, "Improving the Accuracy of Diagnostic Predictions for Power Transformers by Employing a Hybrid Approach Combining Smote and Dnn," *Computers and Electrical Engineering* 117 (2024): 109232.

34. Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature* 521, no. 7553 (2015): 436–444.

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

35. H. K. Dam, T. Pham, S. W. Ng, et al., "Lessons Learned From Using a Deep Tree-Based Model for Software Defect Prediction in Practice," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (IEEE, 2019), 46–57.

36. K. Shi, Y. Lu, G. Liu, Z. Wei, and J. Chang, "MPT-Embedding: An Unsupervised Representation Learning of Code for Software Defect Prediction," *Journal of Software: Evolution and Process* 33, no. 4 (2021): e2330.

37. J. Xu, F. Wang, and J. Ai, "Defect Prediction With Semantics and Context Features of Codes Based on Graph Representation Learning," *IEEE Transactions on Reliability* 70, no. 2 (2020): 613–625.

38. Z. Zhao, B. Yang, G. Li, H. Liu, and Z. Jin, "Precise Learning of Source Code Contextual Semantics via Hierarchical Dependence Structure and Graph Attention Networks," *Journal of Systems and Software* 184 (2022): 111108.

39. M. Cui, S. Long, Y. Jiang, and X. Na, "Research of Software Defect Prediction Model Based on Complex Network and Graph Neural Network," *Entropy* 24, no. 10 (2022): 1373.

40. X. Zhou and L. Lu, "Defect Prediction via Lstm Based on Sequence and Tree Structure," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)* (IEEE, 2020), 366–373.

41. J. Bayer, D. Wierstra, J. Togelius, and J. Schmidhuber, "Evolving Memory Cell Structures for Sequence Learning," in *Artificial Neural Networks–ICANN 2009: 19th International Conference, Limassol, Cyprus, September 14–17, 2009, Proceedings, Part II 19* (Springer, 2009), 755–764.

42. J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," *Neural Networks* 61 (2015): 85–117.

43. Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical Attention Networks for Document Classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Association for Computational Linguistics (ACL), 2016), 1480–1489.

44. X. Fu, J. Yang, J. Li, M. Fang, and H. Wang, "Lexicon-Enhanced Lstm With Attention for General Sentiment Analysis," *IEEE Access* 6 (2018): 71884–71891.

45. G. Lin, J. Zhang, W. Luo, et al., "Cross-Project Transfer Representation Learning for Vulnerable Function Discovery," *IEEE Transactions on Industrial Informatics* 14, no. 7 (2018): 3289–3297.

46. G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software Defect Prediction via Attention-Based Recurrent Neural Network," *Scientific Programming* 2019, no. 1 (2019): 6230953.

47. J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo, "The Jinx on the Nasa Software Defect Data Sets," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering* (Association for Computing Machinery (ACM), 2016), 1–5.

48. M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data Quality: Some Comments on the Nasa Software Defect Datasets," *IEEE Transactions on Software Engineering* 39, no. 9 (2013): 1208–1215.

49. P. Ardimento, L. Aversano, M. L. Bernardi, M. Cimitile, and M. Iammarino, "Just-In-Time Software Defect Prediction Using Deep Temporal Convolutional Networks," *Neural Computing and Applications* 34, no. 5 (2022): 3981–4001.

50. P. Ardimento, L. Aversano, M. L. Bernardi, et al., "Temporal Convolutional Networks for Just-In-Time Software Defect Prediction," in *ICSOFT* (Science and Technology Publications, LDA (SciTePress), 2020), 384–393.

51. R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy, "An Automatically Created Novel Bug Dataset and Its Validation in Bug Prediction," *Journal of Systems and Software* 169 (2020): 110691.

52. K. Phung, E. Ogunshile, and M. E. Aydin, "Domain-Specific Implications of Error-Type Metrics in Risk-Based Software Fault Prediction," *Software Quality Journal* 33, no. 1 (2025): 7.

53. L. Ardito, R. Coppola, L. Barbato, and D. Verga, "A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review," *Scientific Programming* 2020 (2020): 1–26.

54. M. Tan and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *International Conference on Machine Learning* (PMLR, 2019), 6105–6114.

55. T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 1st ed. (Springer, 2001).

56. S. Ioffe, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," arXiv preprint, arXiv:1502.03167 2015.

57. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks From Overfitting," *Journal of Machine Learning Research* 15, no. 1 (2014): 1929–1958.

58. S. Mastromichalakis, "Alrelu: A Different Approach on Leaky Relu Activation Function to Improve Neural Networks Performance," arXiv preprint, arXiv:2012.07564 (2020).

59. P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph Attention Networks," arXiv preprint, arXiv:1710.10903 (2017).

60. Y. Meng, S. Li, X. Wang, C. Liu, and X. Zhang, "Gla-Sdp: A Graph Neural Network and Lstm Based Software Defect Prediction Method With Attention Mechanism," *Journal of Systems and Software* 209 (2025): 111918.

61. K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful Are Graph Neural Networks?," arXiv preprint, arXiv:1810.00826 (2018).

62. D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," arXiv preprint, arXiv:1412.6980 (2014).

63. Z. Zhang, "Improved Adam Optimizer for Deep Neural Networks," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)* (Ieee, 2018), 1–2.

64. S. Ruder, "An Overview of Gradient Descent Optimization Algorithms," arXiv preprint, arXiv:1609.04747 (2016).

65. D. Masters and C. Luschi, "Revisiting Small Batch Training for Deep Neural Networks," arXiv preprint, arXiv:1804.07612 (2018).

66. L. Prechelt, "Early Stopping-But When?," in *Neural Networks: Tricks of the Trade* (Springer, 1998), 55–69.

67. A. Labach, H. Salehinejad, and S. Valaee, "Survey of Dropout Methods for Deep Neural Networks," arXiv preprint, arXiv:1904.13310 (2019).

68. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT press, 2016).

69. F. Pedregosa, G. Varoquaux, A. Gramfort, et al., "Scikit-Learn: Machine Learning in Python," *Journal of Machine Learning Research* 12 (2011): 2825–2830.

70. J. M. Johnson and T. M. Khoshgoftaar, "Survey on Deep Learning With Class Imbalance," *Journal of Big Data* 6, no. 1 (2019): 1–54.

71. W. L. Hamilton, "Graph Representation Learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14, no. 3 (2020): 1–159.

72. R. Kohavi, *A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection*, vol. 14 (IJCAI, 1995), 1137–1145.

73. P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep Reinforcement Learning That Matters," *Proceedings of the AAAI Conference on Artificial Intelligence* 32, no. 1 (2018): 3207–3214.

74. S. Imambi, K. B. Prakash, and G. Kanagachidambaresan, "Pytorch," in *Programming With TensorFlow: Solution for Edge Computing Applications* (Springer, 2021), 87–104.

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

75. M. Fey and J. E. Lenssen, "Fast Graph Representation Learning With Pytorch Geometric," arXiv preprint, arXiv:1903.02428 (2019).

76. O. Kramer and O. Kramer, "Scikit-Learn," in *Machine Learning for Evolution Strategies* (Springer International Publishing, 2016), 45–53.

77. G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-Learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning," *Journal of Machine Learning Research* 18, no. 1 (2017): 559–563.

78. P. Gupta and A. Bagchi, "Introduction to Numpy," in *Essentials of Python for Artificial Intelligence and Machine Learning* (Springer, 2024), 127–159.

79. S. Weisberg, *Yeo-Johnson Power Transformations. Department of Applied Statistics*, vol. 1 (University of Minnesota, 2001), 8.

80. M. Riani, A. C. Atkinson, and A. Corbellini, "Automatic Robust Box–Cox and Extended Yeo–Johnson Transformations in Regression," *Statistical Methods & Applications* 32, no. 1 (2023): 75–102.

81. H. Han, W. Wang, and B. Mao, "Borderline-Smote: A New Over-Sampling Method in Imbalanced Data Sets Learning," in *Advances in Intelligent Computing, 2005* (Springer Berlin Heidelberg, 2005), 878–887.

82. H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive Synthetic Sampling Approach for Imbalanced Learning," in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)* (IEEE, 2008), 1322–1328.

83. G. E. A. P. A. Batista, "A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data," *ACM SIGKDD Explorations Newsletter* 6, no. 1 (2004): 20–29.

84. A. Fernández, S. Garcia, F. Herrera, and N. V. Chawla, "Smote for Learning From Imbalanced Data: Progress and Challenges, Marking the 15-Year Anniversary," *Journal of Artificial Intelligence Research* 61 (2018): 863–905.

85. N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic Minority Over-Sampling Technique," *Journal of Artificial Intelligence Research* 16 (2002): 321–357.

86. M. Gupta, "Software Fault Prediction With Imbalanced Datasets Using Smote-Tomek Sampling Technique and Genetic Algorithm Models," *Multimedia Tools and Applications* 83, no. 16 (2024): 47627–47648.

87. C. Pak, "An Empirical Study on Software Defect Prediction Using Over-Sampling by Smote," *International Journal of Software Engineering and Knowledge Engineering* 28, no. 6 (2018): 811–830.

88. Tamanna, "Feature Reduction Techniques for Software Bug Prediction," in *AIP Conference Proceedings* (American Institute of Physics, 2022).

89. M. Friedman, "A Comparison of Alternative Tests of Significance for the Problem of m Rankings," *Annals of Mathematical Statistics* 11, no. 1 (1940): 86–92.

90. J. Demšar, "Statistical Comparisons of Classifiers Over Multiple Data Sets," *Journal of Machine Learning Research* 7 (2006): 1–30.

91. V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, "A Systematic Review of Effect Size in Software Engineering Experiments," *Information and Software Technology* 49, no. 11–12 (2007): 1073–1086.

92. I. Guyon and A. Elisseeff, "An Introduction to Variable and Feature Selection," *Journal of Machine Learning Research* 3, no. Mar (2003): 1157–1182.

93. S. Saha, "Improving Software Development Using Ai-Enabled Predictive Analytics," *Journal of Artificial Intelligence, Machine Learning & Data Science* 2 (2023): 1050–1053.

94. P. B. Nemenyi, *Distribution-Free Multiple Comparisons* (Princeton University, 1963).

95. M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys* 51, no. 4 (2018): 1–37.

96. Z. Chen, J. Xu, C. Alippi, et al., "Graph Neural Network-Based Fault Diagnosis: A Review," arXiv preprint, arXiv:2111.08185 (2021).

97. J. Cohen, *Statistical Power Analysis for the Behavioral Sciences* (Routledge, 2013).

98. D. Lakens, "Calculating and Reporting Effect Sizes to Facilitate Cumulative Science: A Practical Primer for t-Tests and Anovas," *Frontiers in Psychology* 4 (2013): 863.

99. G. E. Gignac and E. T. Szodorai, "Effect Size Guidelines for Individual Differences Researchers," *Personality and Individual Differences* 102 (2016): 74–78.

100. S. S. Sawilowsky, "New Effect Size Rules of Thumb," *Journal of Modern Applied Statistical Methods* 8, no. 2 (2009): 26.

101. S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code Prediction by Feeding Trees to Transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (IEEE, 2021), 150–162.

102. C. F. Dormann, J. Elith, S. Bacher, et al., "Collinearity: A Review of Methods to Deal With It and a Simulation Study Evaluating Their Performance," *Ecography* 36, no. 1 (2013): 27–46.

103. K. P. Vatcheva, M. Lee, J. B. McCormick, and M. H. Rahbar, "Multicollinearity in Regression Analyses Conducted in Epidemiologic Studies," *Epidemiology (Sunnyvale, Calif.)* 6, no. 2 (2016): 227.

104. J. C. Munson and S. G. Elbaum, "Code Churn: A Measure for Estimating the Impact of Code Change," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (IEEE, 1998), 24–31.

105. A. O. Balogun, S. Basri, S. Mahamad, et al., "Impact of Feature Selection Methods on the Predictive Performance of Software Defect Prediction Models: An Extensive Empirical Study," *Symmetry* 12, no. 7 (2020): 1147.

106. S. Zhang, S. Jiang, and Y. Yan, "A Software Defect Prediction Approach Based on Hybrid Feature Dimensionality Reduction," *Scientific Programming* 2023, no. 1 (2023): 5585130.

107. J. Fox, *Applied Regression Analysis and Generalized Linear Models* (Sage Publications, 2015).

108. I. T. Jolliffe and J. Cadima, "Principal Component Analysis: A Review and Recent Developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374, no. 2065 (2016): 20150202.

109. B. C. Ross, "Mutual Information Between Discrete and Continuous Data Sets," *PLoS One* 9, no. 2 (2014): e87357.

110. Y. Shi, Z. Huang, S. Feng, H. Zhong, W. Wang, and Y. Sun, "Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification," arXiv preprint, arXiv:2009.03509 (2020).

111. R. L. Wasserstein, A. L. Schirm, and N. A. Lazar, "Moving to a World Beyond "p¡ 0.05"," *American Statistician* 73, no. sup1 (2019): 1–19.

112. M. S. Santos, P. H. Abreu, P. J. García-Laencina, A. Simão, and A. Carvalho, "Cross-Validation for Imbalanced Datasets: Avoiding Overoptimistic and Overfitting Approaches," *EURASIP Journal on Advances in Signal Processing* 2018 (2018): 1–16.

113. P. Branco, L. Torgo, and R. P. Ribeiro, "A Survey of Predictive Modeling on Imbalanced Domains," *ACM Computing Surveys* 49, no. 2 (2016): 1–50.

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

114. G. Menardi and N. Torelli, "Training and Assessing Classification Rules With Imbalanced Data," *Data Mining and Knowledge Discovery* 28, no. 1 (2014): 92–122.

115. A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," *Proceedings of the 33rd International Conference on Software Engineering* (2011): 1–10.

116. F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The Role of Replications in Empirical Software Engineering," *Empirical Software Engineering* 13, no. 2 (2008): 211–218.

117. A. Vaswani, N. Shazeer, N. Parmar, et al., *Attention Is All You Need*, vol. 30 (Neural Information Processing Systems (NeurIPS), 2017).

15320634, 2026, 1, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70472 by NICE, National Institute for Health and Care Excellence, Wiley Online Library on [29/06/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License