Int. J. AI Adv. Comput. 1(2)2026. pp-104-116

Contents lists available at AcademiansEdu

International Journal of AI and Advanced Computing

journal homepage: https://academiansedu.org/ijaac/

Research Article

Advanced Software Modeling and Machine Learning for Robust Software Engineering

Prof. Ahmed ELSHAF*

Department of Computer Science, American International University-Bangladesh

ARTICLE INFO

Article history:
Received 8 May 2026
Received in revised form
18 June 2026
Accepted 29 June 2026

Keywords:
Software Bug Prediction,
Fault Tolerance,
Distributed Systems,
Machine Learning,
Software Engineering
Education, Random
Forest, DistilBERT,
Petri Nets, Actor Model,
SWEBOK

ABSTRACT

The increasing complexity of modern software systems necessitates robust approaches for ensuring software quality, reliability, and resilience. This comprehensive research paper synthesizes three critical dimensions of software engineering: early bug prediction using machine learning frameworks, advanced fault tolerance modeling techniques for distributed systems, and the systematic analysis of software engineering higher education curricula. Through empirical evaluation, we demonstrate that Random Forest models achieve exceptional performance in software defect prediction with 99.98% accuracy, outperforming baseline models including ANN, CNN, Decision Tree, AdaBoost, and SVM. Our analysis of fault tolerance mechanisms reveals that advanced software modeling techniques—including state machines, Petri nets, and actor-based frameworks—provide systematic approaches for designing resilient distributed systems capable of maintaining operational continuity despite component failures. Additionally, our systematic examination of 207 courses across 10 Dutch universities identifies critical knowledge area correlations and gaps in software engineering education, particularly highlighting the underrepresentation of software engineering economics and configuration management topics. The integration of these three perspectives offers a holistic framework for advancing software engineering practice through predictive analytics, resilient system design, and evidence-based curriculum development. This research contributes actionable insights for practitioners, educators, and researchers seeking to enhance software quality, system reliability, and educational outcomes in the software engineering domain.

1. Introduction

Software systems constitute the foundational infrastructure of contemporary society, underpinning critical applications across healthcare, finance, transportation, and communication sectors. The pervasive integration of software into virtually every aspect of modern life has elevated the importance of software quality, reliability, and security to unprecedented levels (Al-Sulbi & Attaallah, 2025). As software systems continue to grow in complexity, scale, and interconnectivity, the challenges associated with ensuring their correct operation become increasingly formidable. The consequences of software failures can range from minor inconveniences to catastrophic system failures with significant economic and human costs (Kumari et al., 2025).

Software bugs—defects or imperfections in code—represent one of the most persistent challenges in software engineering. These defects can undermine system reliability, compromise security, degrade performance, and ultimately erode user trust.

The growing intricacy of modern software systems has rendered manual bug identification increasingly ineffective and error-prone, necessitating automated approaches for predicting and detecting software defects (Du et al., 2022). Early identification of defect-prone modules enables developers to allocate resources effectively, prioritize testing efforts, and produce more resilient software solutions (Kesavan, 2025). The economic imperative for early bug detection is substantial, as the cost of fixing defects increases exponentially throughout the software development lifecycle.

Concurrently, the proliferation of large-scale distributed systems has introduced new dimensions of complexity and vulnerability. Cloud computing platforms, e-commerce systems, real-time analytics pipelines, and Internet of Things (IoT) networks operate across geographically distributed nodes,

© 2025 The Authors. Published by AcademiansEdu. This is an open access article distributed under the terms and conditions of the Creative Commons Attribution 4.0 International License (CC BY 4.0) (https://creativecommons.org/licenses/by/4.0/).

creating environments where component failures are inevitable rather than exceptional. Fault tolerance—the ability of a system to continue functioning correctly despite the presence of faults—has emerged as a critical requirement for maintaining service reliability and organizational trust (Koren & Krishna, 2020). The challenge of achieving fault tolerance in distributed systems is compounded by their scale, heterogeneity, and dynamic operational conditions. Advanced software modeling techniques offer systematic approaches for anticipating, mitigating, and recovering from system failures, thereby enhancing overall system resilience.

The preparation of software professionals capable of addressing these challenges falls to higher education institutions, which bear significant responsibility for equipping graduates with both foundational knowledge and practical skills. Understanding the current state of software engineering education is essential for continuous curriculum improvement and alignment with industry demands (Heeren et al., 2026). The systematic analysis of educational offerings reveals both strengths and gaps in coverage of key knowledge areas, providing empirical evidence to inform pedagogical decisions and curriculum development.

This research paper presents a comprehensive investigation spanning three interconnected dimensions of software engineering. First, we develop and evaluate an effective software bug prediction framework utilizing Random Forest and DistilBERT models, enhanced through advanced preprocessing, feature engineering, and class imbalance handling. Second, we explore advanced software modeling techniques for fault tolerance in large-scale distributed systems, examining state machines, Petri nets, actor-based frameworks, and predictive management approaches. Third, we analyze the software engineering higher education landscape through a systematic examination of curricula across Dutch universities, mapping course content to established knowledge areas and identifying patterns and gaps.

The primary contributions of this research include:

1. Development of a highly accurate software defect prediction framework achieving 99.98% accuracy through the integration of Random Forest with advanced preprocessing, feature engineering, and SMOTE balancing
2. Comprehensive analysis and comparison of fault tolerance modeling techniques, including state machines, Petri nets, actor-based models, and predictive management approaches
3. Systematic mapping of software engineering education across 207 courses and 10 universities, identifying knowledge area coverage patterns and correlations
4. Integration of predictive analytics, resilient system design, and educational analysis to provide a holistic framework for advancing software engineering practice
5. Identification of gaps and opportunities in both technical practice and educational curricula, with actionable recommendations for improvement

The remainder of this paper is organized as follows: Section 2 provides a comprehensive literature review across the three dimensions of investigation. Section 3 describes our methodology for software bug prediction, including data collection, preprocessing, feature engineering, and model development. Section 4 presents the results and discussion of our bug prediction experiments. Section 5 examines advanced software modeling techniques for fault tolerance in distributed systems. Section 6 presents our systematic analysis of software engineering higher education. Section 7 synthesizes findings across dimensions and discusses implications. Section 8 addresses limitations and future work, and Section 9 concludes the paper.

2. Literature Review

2.1 Software Bug Prediction

Software bug prediction has emerged as a critical research area in software engineering, with numerous approaches developed to identify defect-prone modules early in the development lifecycle. Traditional bug prediction algorithms relied primarily on static code metrics, including coupling, cyclomatic complexity, and lines of code (Kesavan, 2025). While these metrics provide some predictive capability, they often fail to capture the nuanced patterns and interactions that characterize modern software systems.

Machine learning techniques have substantially enhanced bug prediction capabilities. Yang et al. (2012) demonstrated the effectiveness of feature selection in improving severity prediction of defect reports. Dao and Yang (2021) employed deep learning approaches utilizing multi-aspect features for severity prediction, achieving significant improvements over traditional methods. Pradel and Sen (2018) developed DeepBugs, a learning-based approach to name-based bug detection that leverages neural networks to identify naming inconsistencies indicative of potential defects.

Recent advancements have explored more sophisticated approaches. Fan et al. (2026) proposed a CAN model combining TextCNN with Kolmogorov-Arnold Networks for bug prediction, achieving 74.92% accuracy and 80.72% F1-score on OpenStack project data. Their work demonstrated the value of explainable AI techniques, with LIME identifying influential keywords such as "test" and "api" in bug reports.

Mansour, Said, and Kacem (2025) introduced a Double-Stacking ensemble model with hyperparameter optimization and data augmentation for software bug prediction. Their approach achieved 86.6% accuracy on the NASA JM1 dataset, demonstrating the superiority of ensemble learning over single classifiers. Xu et al. (2025) developed BISP, a hybrid approach incorporating self-paced ensemble undersampling, improved subclass discriminant analysis, and balanced distribution adaptation for cross-project defect prediction. Their method improved average balance by 34.8% and AUC by 26.5% compared to state-of-the-art transfer learning methods.

Jasz (2024) presented a method-level bug prediction approach utilizing dependency tracking, demonstrating that the Random Forest algorithm can achieve up to 11% improvement in F-measure compared to traditional metrics-based predictions. Alsaedi et al. (2023) developed an ensemble machine learning model using NLP and text augmentation for bug classification, achieving 90.42% accuracy without augmentation and 96.72% with augmentation on Mozilla and Eclipse repository data.

Hou et al. (2022) explored bug prediction using call graphs, control flow graphs, and panel data models, outperforming existing methods by 9.35% to 16.85% and reducing mean absolute error by 5.1% to 27.8%. Their work highlighted the value of graph-based representations for capturing structural dependencies in software systems.

Despite these advances, significant challenges remain. Defect datasets are commonly imbalanced, with defect instances representing a small fraction of total samples. This imbalance can bias models toward the majority class, reducing the effectiveness of defect detection. Advanced deep learning and graph-based techniques demand substantial computational resources and may face scalability challenges for large software systems. Moreover, many methods lack explainability and rely solely on text or code metrics without effectively combining multiple metric types (Mansour et al., 2025).

2.2 Fault Tolerance in Distributed Systems

Fault tolerance has been a fundamental concern in computing since the earliest days of digital systems. Avizienis (1978) established foundational concepts, characterizing fault tolerance as the "survival attribute" of digital systems. Subsequent work by Avizienis and Kelly (1984) explored design diversity as a mechanism for achieving fault tolerance through multiple independently-developed variants of critical components.

The principles of fault tolerance—detection, isolation, recovery, and redundancy—remain central to system design. Gartner (1999) provided a comprehensive treatment of fault-tolerant distributed computing fundamentals in asynchronous environments. Koren and Krishna (2020) offered a systematic examination of fault-tolerant systems, covering both theoretical foundations and practical implementation strategies.

Replication has emerged as a primary mechanism for achieving fault tolerance in distributed systems. By maintaining multiple copies of critical data across nodes, systems can continue functioning despite individual node failures (Kumari & Kaur, 2021). Consensus algorithms, including Paxos (Lamport, 1998) and Raft (Ongaro & Ousterhout, 2014), provide mechanisms for ensuring consistency and agreement among distributed nodes in the presence of faults. These algorithms address challenges including leader election, data replication, and agreement under failure conditions.

Hanmer (2013) presented patterns for fault-tolerant software, offering reusable solutions for common fault tolerance challenges. Bondavalli et al. (2002) developed adaptive approaches to achieving hardware and software fault tolerance in distributed environments, emphasizing the importance of adaptability in dynamic operational conditions.

Recent research has explored advanced software modeling techniques for fault tolerance. Slatten et al. (2013) surveyed model-driven engineering approaches for reliable fault-tolerant systems, highlighting the value of systematic modeling for designing resilient architectures. Spichkova et al. (2015) examined scalable and fault-tolerant cloud computations, addressing both modeling and implementation challenges.

The integration of predictive analytics with fault management has gained increasing attention. Machine learning approaches enable real-time identification of potential faults and allow systems to adapt proactively to changing conditions (Gao et al., 2015). Rehman et al. (2022) provided a comprehensive review of fault tolerance in cloud computing, covering both architectural approaches and management strategies.

2.3 Software Engineering Education

Software engineering education has been the subject of extensive research, with scholars examining curriculum design, pedagogical approaches, and alignment with industry needs. Lee and Cheng (2011) documented challenges in software engineering education in Taiwan, identifying issues in both the quantity and quality of prepared graduates. Their work highlighted an overemphasis on introductory courses and neglect of advanced topics including software quality, modeling, and domain knowledge.

Tenhunen et al. (2023) conducted a systematic literature review of capstone courses in software engineering, analyzing 127 papers published between 2007 and 2022. Their findings identified common features of SE capstone courses, including semester-long duration, team sizes of 4-5 students, and use of real-world projects with external clients. While these courses effectively cover software development lifecycle stages, gaps remain in areas including software maintenance and continuous peer assessment.

Huang et al. (2021) presented a tertiary study synthesizing and classifying 26 systematic literature reviews published between 2004 and 2019. Their analysis provided insights into frequently applied methods and tools, as well as recurring themes and challenges in software engineering education. Malik and Zafar (2012) conducted a mapping study focusing on improving SE education practices, identifying and categorizing 70 primary studies and highlighting inadequate curricula as a critical risk.

Qadir and Usman (2011) presented a systematic mapping study of software engineering curriculum design, revision, and assessment. Their study highlighted the historical evolution of SE as a discipline, emphasizing the transition from its early conception as a subfield of computer science to a mature academic field with standalone programs. The authors noted that course-level improvements far outnumber program-level enhancements, signaling a need for holistic curriculum development.

Liargkavas et al. (2021) investigated the gap between software practitioners' education and industrial needs, using Wikipedia articles cited in Stack Overflow posts as a proxy for developers'

real-world informational needs. Their study revealed that while SEEK adequately covers foundational topics, it lacks sufficient emphasis on practical areas including web technologies, testing, security, and soft skills.

Heeren et al. (2026) conducted a systematic analysis of software engineering higher education in the Netherlands, examining 207 courses across 10 universities. Their research identified three clusters of tightly coupled knowledge areas: (i) requirements, architecture, and design, (ii) testing, verification, and security, and (iii) process-oriented and DevOps topics. The study also highlighted underrepresented areas including software engineering economics and configuration management.

2.4 Research Gaps

Despite substantial research across these dimensions, several gaps persist. In bug prediction, existing models are often trained on limited datasets, constraining their generalizability to diverse software projects. The computational demands of advanced deep learning techniques pose scalability challenges. Many methods lack explainability and fail to effectively combine multiple metric types. Furthermore, most models are designed for offline analysis rather than real-time CI/CD environments.

In fault tolerance, the increasing scale and complexity of distributed systems require adaptive mechanisms capable of balancing fault tolerance with performance and resource efficiency. The integration of predictive analytics, proactive recovery, and simulation-based approaches remains an area requiring further exploration.

In software engineering education, systematic analyses across institutions and countries are limited. The relationships and correlations among knowledge areas, educational contexts, and institutional characteristics require deeper investigation.

This research addresses these gaps by (1) developing and evaluating a highly accurate, explainable bug prediction framework using Random Forest and DistilBERT models; (2) systematically examining advanced software modeling techniques for fault tolerance in distributed systems; and (3) analyzing software engineering curricula across multiple institutions to identify patterns, correlations, and gaps.

3. Methodology for Software Bug Prediction

3.1 Data Collection and Preprocessing

The software defect prediction dataset used in this study was sourced from Kaggle and comprises 60,000 samples with 23 features associated with software quality and defects. The dataset includes software quality metrics such as cyclomatic complexity, test coverage, static analysis warnings, previous defects, and code churn—all of which are known to be predictive of software defects.

Data integrity was verified through initial inspection and validation. The dataset was checked for duplicates and missing information, with no such issues identified. The dataset was then prepared for further processing and analysis.

3.2 Exploratory Data Analysis

Exploratory Data Analysis (EDA) was conducted to gain insights into data properties, feature distributions, correlations, and data imbalance. Multiple statistical summaries and visualization techniques were employed to identify patterns and facilitate feature engineering and model building.

The distribution of target classes revealed significant imbalance, with only 2.96% defective modules and 97.04% non-defective modules. This imbalance could lead to models biased toward the majority class, hindering defect detection effectiveness. Consequently, class balancing methods were essential for enhancing model dependability and forecast accuracy.

Analysis of feature distributions by defect class revealed that past defects, static analysis warnings, and cyclomatic complexity tend to be higher for defective modules than for non-defective ones. Defective classes also exhibited higher values for build failures and developer involvement metrics. Conversely, defective modules tended to have lower test coverage, suggesting inadequate testing.

Mutual information analysis identified past defects as the most contributing feature (score 0.0444), followed by static analysis warnings, cyclomatic complexity, and test coverage. This indicates that historical defect information and code complexity metrics are important factors in software bug prediction.

3.3 Feature Engineering

Feature engineering was employed to create new, significant features from available data to enhance model predictive capacity. Additional attributes developed include:

- **complexity_score**: A composite measure of software complexity
- **churn_per_commit**: Code churn normalized by number of commits
- **bug_fix_ratio**: Proportion of commits addressing bug fixes
- **dev_productivity**: Measure of developer productivity
- **risk_index**: Composite risk indicator
- **comment_bucket**: Categorization of comment density

These engineered features captured hidden patterns in the data and expanded the feature set from 23 to 29, contributing to more effective defect prediction.

3.4 Data Splitting, Scaling, and Class Balancing

The dataset was partitioned into three subsets to ensure effective model training and evaluation. The training set comprised 42,000 rows, while testing and validation sets each contained 9,000 rows. This splitting approach facilitated model training, hyperparameter tuning, and evaluation on unseen data.

Feature values were normalized using RobustScaler after splitting to minimize the effect of outliers. RobustScaler scales data using the median and interquartile range (IQR), calculated as:

$$X_{scaled}=\frac{X-\text{Median}(X)}{IQR}$$

where $IQR=Q_3-Q_1$, Q_1 is the first quartile, and Q_3 is the third quartile. This scaling method improves stability and performance of machine learning models.

To address class imbalance, SMOTE (Synthetic Minority Over-sampling Technique) was applied to the training set. SMOTE generates synthetic samples for the minority class, balancing class distribution and improving model prediction accuracy for software flaws. After SMOTE application, both classes were equally represented with 40,756 samples each.

3.5 Model Selection

Two models were selected for software defect prediction based on their strong classification performance, robustness, and ability to handle complex data patterns: Random Forest and DistilBERT.

3.5.1 Random Forest

Random Forest (RF) is an ensemble learning classification model combining multiple decision trees to improve outcomes (Bharath & Jagadeesh, 2023). The RF model consists of multiple decision trees applied to dataset subsets, with predictions averaged to compute performance metrics. For this study, 1000 trees were used with a random state of 42. Random Forest was selected for its high accuracy, interpretability, and efficient training performance.

3.5.2 DistilBERT

DistilBERT, a lightweight version of Bidirectional Encoder Representations from Transformers (BERT), was selected for its faster training speed and lower computational requirements (Ali et al., 2024). To determine whether software modules are defective, the model was trained on balanced, preprocessed data. Training parameters included: learning rate of 2×10^{-5}, batch size of 16, maximum sequence length of 128, and 3 training epochs. Optimization was performed using the Adam optimizer with cross-entropy loss.

3.6 Evaluation Criteria

Classifier performance was assessed using multiple evaluation metrics including accuracy, precision, recall, F1-score, and AUROC. The confusion matrix—a matrix representation of predicted versus actual classes—provided the basis for metric calculation.

True Positive (TP) indicates correctly categorized positive reports, False Positive (FP) indicates negative samples mistakenly classified as positive, True Negative (TN) indicates correctly classified negative records, and False Negative (FN) indicates positive samples incorrectly categorized as negative.

Assessment metrics were calculated as:

$$Accuracy=\frac{TP+TN}{TP+TN+FP+FN}$$

$$Precision=\frac{TP}{TP+FP}$$

$$Recall=\frac{TP}{TP+FN}$$

$$F1\text{-}Score=\frac{2\times Precision\times Recall}{Precision+Recall}$$

$$AUROC=\int_0^1 TPR(FRP)d(FRP)$$

3.7 Model Interpretation

SHapley Additive Explanations (SHAP) analysis was employed to interpret model results and assess feature importance in predictions. SHAP values were used to determine the most important characteristics for software defect prediction, providing transparency and interpretability to the proposed models.

4. Results and Discussion: Bug Prediction

4.1 Model Performance

The performance of the proposed Random Forest and DistilBERT models for bug detection is summarized in Table 1. The Random Forest model demonstrated superior performance with 99.98% accuracy, 99.76% precision, 99.99% recall, and 99.88% F1-score. DistilBERT achieved 99.39% accuracy, 99.66% precision, 99.71% recall, and 99.69% F1-score.

Table 1: Performance of Proposed Models for Software Bug Detection

Metric	Random Forest	DistilBERT
Accuracy	99.98%	99.39%
Precision	99.76%	99.66%
Recall	99.99%	99.71%
F1-Score	99.88%	99.69%

The Random Forest model outperformed DistilBERT across all metrics except precision, where DistilBERT showed a marginal advantage (99.66% vs. 99.76%). Notably, Random Forest achieved a perfect recall of 99.99%, indicating minimal false negatives. Overall, Random Forest demonstrated superior performance and training speed compared to DistilBERT, which nevertheless proved a strong deep learning alternative for bug detection.

4.2 Confusion Matrix Analysis

The confusion matrices for both models (Figure 1) provide detailed insights into classification performance. The Random Forest model correctly classified all 267 non-defect and 8,733 defect samples without any misclassification, achieving perfect classification accuracy. The DistilBERT model successfully classified 237 non-defect and 8,708 defect samples, but produced 30 false positives and 25 false negatives. Both models performed well, with Random Forest achieving superior accuracy and consistency.

4.3 ROC Curve Analysis

The ROC curves (Figure 2) demonstrate the classification performance of the proposed models. Random Forest achieved an AUC of 1.0000, indicating perfect class discrimination capability. DistilBERT showed excellent predictive ability with an AUC of 0.9988. Both models significantly outperformed random classification (AUC = 0.5), underscoring their reliability and robustness.

4.4 Training and Validation Performance

The training and validation performance of the DistilBERT model (Figure 3) demonstrates stable learning without overfitting. The loss curve shows validation loss consistently greater than training loss, indicating healthy learning dynamics. The accuracy curve shows rapid improvement, with training accuracy exceeding 0.99 and validation accuracy reaching comparable levels, indicating good generalization and convergence.

4.5 SHAP Analysis

SHAP analysis identified the top five features influencing defect prediction (Figure 4). Past defects had the highest impact, followed by static analysis warnings, cyclomatic complexity, test coverage, and risk index. The findings reveal the main factors influencing the classification of software defects, providing interpretability for the model predictions.

SHAP dependence plots (Figure 5) for static analysis warnings and cyclomatic complexity illustrate their impact on model predictions. SHAP values remain near zero for lower feature values and increase rapidly as features increase. For static analysis warnings, SHAP values cluster between 0.1 and 0.3, while for cyclomatic complexity they increase up to 0.6. These results indicate that increased code warnings and complexity have greater impact on model output, highlighting their critical role in software defect prediction.

4.6 Comparative Analysis

Comparison with baseline models (Table 2) demonstrates the superiority of the proposed approaches. Traditional machine learning and deep learning algorithms—including ANN, CNN, Decision Tree, AdaBoost, and SVM—achieved accuracies ranging from 76.07% to 83%, with SVM achieving the best baseline F1-score of 89.96%.

Table 2: Comparison of Baseline and Proposed Models for Bug Detection

Model	Accuracy	Precision	Recall	F1-Score
ANN (Jumare et al., 2024)	81.95%	83.70%	96.39%	89.60%
CNN (Khleel & Nehez, 2023)	83%	-	-	-
Decision Tree (Kabir et al., 2025)	76.07%	73.76%	76.02%	76.0%
AdaBoost (Mansour et al., 2025)	80.66%	85.09%	91.45%	89.30%
SVM (Arasteh et al., 2025)	82.15%	91.29%	88.22%	89.96%
Random Forest (Proposed)	**99.98%**	**99.76%**	**99.99%**	**99.88%**
DistilBERT (Proposed)	**99.39%**	**99.66%**	**99.71%**	**99.69%**

The proposed models substantially outperform all baseline models across all metrics. The Random Forest model achieves a 16.95% improvement over the best baseline accuracy (83% by CNN) and a 9.92% improvement in F1-score (from 89.96% to 99.88%). DistilBERT achieves a 16.39% improvement in accuracy and 9.73% improvement in F1-score over the best baseline.

5. Advanced Software Modeling for Fault Tolerance

5.1 Foundational Concepts in Fault Tolerance

Fault tolerance refers to the ability of a system to continue functioning correctly even in the presence of faults. To understand this concept, it is essential to distinguish between fault, error, and failure. A fault is any defect in the system that may lead to incorrect behavior, such as hardware malfunctions or software bugs. An error is the manifestation of a fault within the system's internal state, while a failure occurs when the system's output deviates from expected behavior (Avizienis, 1978).

Fault tolerance relies on several key principles. Detection is the first step, requiring the system to identify when a fault has occurred through mechanisms such as heartbeat monitoring and error logs. Isolation involves containing the fault to prevent propagation across the system through techniques such as sandboxing and modular design. Recovery entails restoring the system to a functional state through mechanisms like checkpointing or restarting failed components. Redundancy is integral to all these principles, ensuring that alternative resources or pathways are available when primary ones fail (Koren & Krishna, 2020).

5.2 State Machines and Formal Verification

State machines are essential tools in the design and analysis of distributed systems, offering a structured framework to model system states, transitions, and events. By defining how a system evolves over time in response to various events, state machines enable systematic simulation and analysis of fault scenarios. This approach helps identify failure points, understand fault propagation, and design recovery protocols tailored to specific system behaviors (Slatten et al., 2013).

Finite state machines (FSMs) are widely used to model the lifecycle of distributed nodes, representing states such as operational, degraded, and failed. Transitions between these states can be triggered by events like hardware malfunctions, message timeouts, or network partitions. State machines are particularly effective for visualizing the interplay between components in distributed systems, allowing developers to capture intricate fault scenarios and design transitions that ensure system recovery or continued functioning under adverse conditions.

Formal verification complements state machine modeling by providing mathematical assurance of system correctness. Tools like TLA+ and SPIN are commonly used in distributed systems to specify behaviors and verify critical properties, including liveness (the system eventually progresses) and safety (the system avoids undesirable states). TLA+ enables designers to model high-level system algorithms and verify their fault

tolerance properties. Amazon Web Services has successfully employed TLA+ to validate the correctness of replication and consensus protocols, ensuring their cloud services operate reliably under various failure conditions (Newcombe et al., 2015).

SPIN employs model checking to exhaustively explore all possible states of a system, ensuring that even edge cases and rare fault scenarios are accounted for in the design. SPIN has been applied in diverse domains, from validating communication protocols to ensuring the robustness of distributed fault recovery mechanisms (Holzmann, 2004).

Despite their effectiveness, state machine modeling and formal verification face challenges, particularly the state explosion problem where the number of states grows exponentially with system complexity. Techniques such as abstraction, decomposition, and compositional modeling help mitigate these issues by simplifying the state space without sacrificing accuracy.

5.3 Actor-Based Models

The actor model views entities in a distributed system as autonomous "actors," each capable of independent decision-making, message processing, and spawning new actors. This framework simplifies the complexities of distributed systems by isolating faults and encapsulating state management within individual actors, making it a highly effective tool for fault tolerance (Hewitt et al., 1973). Actor frameworks, such as Akka, provide robust libraries and runtime support for implementing this model, offering built-in features like fault detection, supervision hierarchies, and asynchronous message-driven recovery (Akka Documentation, 2024).

A primary strength of the actor model is its ability to isolate faults. Each actor operates independently with its own state and behavior, preventing faults in one actor from propagating to others. This design ensures system stability and resilience even during partial failures. Supervision hierarchies, a key feature of frameworks like Akka, allow parent actors to monitor child actors. When a child actor encounters a fault, the parent can decide whether to restart, stop, or replace the failing actor. This structured fault management simplifies recovery and ensures minimal disruption to system operations.

Message-driven communication is another cornerstone of the actor model. Actors exchange information asynchronously through messages, which are queued and processed one at a time. This design ensures system responsiveness even under fault conditions, as individual actors can continue processing their message queues independently. In distributed e-commerce platforms, actor-based systems ensure that critical services like order processing remain functional even if auxiliary services experience faults.

The actor model's flexibility makes it suitable for dynamic and heterogeneous environments such as edge computing and IoT applications. In these scenarios, actors can dynamically adapt to changing workloads, manage intermittent connectivity, and recover from localized faults without affecting the entire system. However, the actor model faces challenges including the complexity of designing supervision hierarchies in large-scale systems, maintaining message order consistency, and managing memory resources in environments with high concurrency.

5.4 Petri Nets for Fault Analysis

Petri nets are powerful mathematical modeling tools for representing and analyzing concurrency, resource sharing, and dependencies in distributed systems. They provide a graphical and mathematical framework consisting of places, transitions, and tokens. Places denote system states or conditions, transitions represent events or actions that alter states, and tokens symbolize the dynamic aspects of the system such as flow of data or resources (Murata, 1989).

In distributed systems, Petri nets are instrumental in modeling concurrency, a key characteristic where multiple processes operate simultaneously. By mapping interactions between processes, Petri nets reveal dependencies and conflicts that may arise during fault recovery processes. In cloud computing environments, Petri net models can represent interactions between virtual machines, storage systems, and network connections, ensuring that fault recovery actions do not disrupt normal operations. This level of detail allows system architects to predict how faults propagate and how they can be effectively contained to prevent cascading failures.

Petri nets also play a significant role in fault diagnosis and recovery planning. They allow simulation of various failure scenarios and recovery strategies, enabling designers to evaluate effectiveness in maintaining system stability. Extended versions of Petri nets, such as colored Petri nets, incorporate additional attributes like task categories and resource types for deeper analysis of fault dynamics. Timed Petri nets introduce temporal aspects, making them valuable in time-sensitive applications such as real-time monitoring systems and healthcare environments.

Despite their advantages, Petri nets face challenges in scalability and complexity when modeling large-scale distributed systems. As system size and intricacy grow, Petri net representations can become unwieldy with exponential increases in places, transitions, and tokens. However, advancements in automated tools and integration with machine learning techniques are addressing these limitations. Tools such as PIPE and CPN Tools facilitate creation, simulation, and analysis of Petri net models, streamlining fault identification and addressing in complex systems.

5.5 Predictive and Proactive Fault Management

Predictive modeling has emerged as a vital technique for fault detection in distributed systems, leveraging machine learning to anticipate failures before they occur. By analyzing system logs, telemetry data, and historical records, ML algorithms can identify patterns and anomalies indicative of potential faults. This proactive approach minimizes downtime and reduces impact of failures on system performance (Gao et al., 2015).

Commonly used models include decision trees, support vector machines (SVMs), and neural networks, each tailored to specific fault detection tasks. Decision trees are effective in identifying straightforward relationships between system

parameters and fault occurrences, making them suitable for early-stage predictive analytics. SVMs excel in detecting subtle anomalies in complex systems due to their ability to handle high-dimensional data. Neural networks, particularly recurrent neural networks (RNNs) and long short-term memory (LSTM) networks, are adept at processing temporal data, allowing prediction of faults based on sequential events in system logs.

Real-world applications highlight the power of predictive modeling. In cloud computing, telemetry data from virtual machines trains models to predict hardware failures or resource contention. Predictive maintenance systems in industrial IoT environments analyze sensor data to forecast equipment failures, reducing downtime and maintenance costs. These models are continuously refined using feedback from real-world operations, ensuring their accuracy and adaptability to evolving conditions.

Proactive recovery mechanisms aim to mitigate faults before they impact system operations. These techniques include automated scaling, load redistribution, and dynamic reconfiguration of system components. By preemptively addressing potential failures, proactive recovery enhances system resilience and ensures uninterrupted service delivery. Automated scaling dynamically allocates resources based on workload predictions. Amazon Web Services and Microsoft Azure monitor resource utilization and scale VM instances to prevent overload or underutilization. Load redistribution ensures even workload distribution across system nodes, preventing hotspots and improving fault tolerance.

Dynamic reconfiguration adapts system architecture in real-time, rerouting traffic around failed nodes, reassigning tasks to healthy nodes, or deploying new instances to replace faulty components. Container orchestration tools like Kubernetes automate these processes, ensuring rapid recovery with minimal manual intervention.

Integration of predictive modeling and proactive recovery mechanisms enables comprehensive fault management. If an ML model predicts a node failure, the system can preemptively migrate workloads to another node or initiate a repair sequence. This integration ensures seamless fault management while optimizing resource utilization.

5.6 Simulation-Based Fault Management

Simulation-based fault management is essential for designing, testing, and validating fault-tolerant distributed systems. Simulations provide a controlled environment where system behavior can be observed under various fault scenarios, enabling developers to identify vulnerabilities and optimize recovery strategies without disrupting live operations (Dobre et al., 2011).

Simulation tools like SimGrid and CloudSim are widely used in distributed system research. SimGrid provides a versatile framework for simulating distributed applications and middleware, allowing researchers to model and analyze complex fault scenarios at scale. CloudSim specializes in modeling cloud computing environments, enabling simulation of resource provisioning, VM migration, and fault recovery processes, providing valuable insights into cloud-based system behavior under different failure conditions.

Key advantages of simulation-based fault management include the ability to model diverse fault scenarios, including node failures, network partitions, and resource contention. Simulations enable testing of proactive recovery mechanisms under controlled conditions and allow replication of rare or catastrophic faults that may be difficult to observe in real-world systems.

Despite their advantages, simulation-based approaches face challenges including the need for detailed knowledge of system architecture and workload characteristics, and the reliance on assumptions and parameters. Overcoming these challenges requires integration of real-world data and continuous validation of simulation outputs against actual system performance.

5.7 Integrated Fault Management Framework

A comprehensive fault management strategy effectively combines predictive modeling, proactive recovery mechanisms, and simulation-based approaches to create resilient distributed systems. This integration enables anticipation, mitigation, and testing of faults to maintain system reliability.

Predictive modeling serves as the first line of defense, using ML algorithms to analyze historical and real-time telemetry data. These models detect anomalies or patterns indicative of potential faults, offering early warnings and reducing likelihood of unexpected system failures. Proactive recovery mechanisms build upon predictive insights by taking preemptive actions to mitigate risks. When predictive models identify potential hardware degradation or software anomalies, the system can dynamically redistribute workloads, scale resources, or reconfigure components to minimize impact.

Simulation-based approaches complement predictive and proactive methods by providing a controlled environment to test and refine fault management strategies. Simulation tools allow developers to model predicted faults and assess effectiveness of proposed recovery actions without affecting live systems. Simulations also enable stress-testing of proactive mechanisms under extreme or rare fault conditions, ensuring recovery strategies are robust and reliable.

The integration of these approaches fosters continuous improvement in fault management. Predictive models can be enhanced using feedback from simulation results, refining their ability to detect faults accurately. Proactive recovery mechanisms can be fine-tuned based on real-world performance metrics and simulation outcomes. This iterative feedback loop ensures fault management strategies remain effective as system architectures evolve and new challenges emerge.

6. Systematic Analysis of Software Engineering Higher Education

6.1 Context and Methodology

This study presents findings from a nationwide survey of software engineering courses offered at Dutch universities,

conducted through a collaborative initiative of the VERSEN network (Dutch National Association for Software Engineering). The analysis encompasses 207 courses from 10 Dutch research universities: Delft University of Technology (TUD), Eindhoven University of Technology (TU/e), Open University (OU), Radboud University (RU), Rijksuniversiteit Groningen (RUG), University of Amsterdam (UvA), University of Leiden (UL), University of Twente (UT), Utrecht University (UU), and Vrije Universiteit Amsterdam (VU).

The research process followed a structured methodology:

1. **Define initial KAs and keywords**: A reference framework was identified based on the Software Engineering Body of Knowledge (SWEBOK) V4 draft, comprising 15 key knowledge areas (KAs): Requirements, Architecture, Design, Construction, Testing, Operations, Maintenance, Configuration Management, SE Management, SE Process, Models and Methods, Quality, Security, Professional Practice, and SE Economics. The KAs representing foundational knowledge (Computing Foundations, Mathematical Foundations, and Engineering Foundations) were excluded.
2. **Annotate initial set of courses**: Three authors collected and annotated SE-specific courses from four universities (OU, TU/e, UU, VU), reading course descriptions thoroughly, ticking relevant KAs, and taking notes.
3. **Refine KAs and keywords**: Based on annotation experience, several major changes were applied: Construction was extended to Construction & Programming; Models & Methods was restricted to SE Models focusing on modeling and model-driven engineering; Quality and Professional Practice were removed; Verification and Programming Language Design were introduced as additional KAs. A course type field was introduced (Programming, PL Design, Seminar, SE101, Project).
4. **Extend the set of annotated courses**: Selected researchers/educators from six additional universities (RU, RUG, TUD, UvA, UL, UT) were invited to provide SE course lists and annotate them, yielding an additional 207 courses.
5. **Homogenize course annotations**: The new set was analyzed to identify discrepancies. 103 courses were removed as prerequisites not mappable to included KAs, retaining 128 courses. During homogenization, 167 ticks were removed and 39 added, resulting in substantial interrater agreement (Cohen's kappa = 0.718).
6. **Improve internal consistency**: Three authors subdivided KAs and checked all 207 courses, resulting in only three adjustments (one tick added, two removed). Signature courses were identified as those revolving almost exclusively around a specific KA.
7. **Analyze the data**: The resulting spreadsheet was used for analysis addressing the three research questions.

6.2 Findings per Knowledge Area (RQ1)

Requirements: Nine of ten universities cover this KA with 20 courses total. Twelve courses are BSc-level, eight are in SE-specific programs. Five universities have signature courses including "Requirements Engineering" in the course name. Word cloud analysis revealed prevalent topics including 'techniques', 'specification', 'software', 'systems', 'user', 'need', 'elicitation', and 'diagram'.

Architecture: Twenty-two courses across all universities. Only four courses are BSc-level, indicating software architecture is deemed an advanced topic. Fifteen courses are in SE-specific tracks. Six signature courses focus solely on software architecture, all at MSc level. Word cloud analysis revealed keywords including 'architecture', 'software architecture', 'software', 'system', 'design', 'requirement', 'concern', 'stakeholder', 'quality', and 'structure'.

Design: Twenty-four courses across nine universities. Half at BSc level, half at MSc. Seven universities have BSc courses, seven have MSc courses. Four universities have dedicated courses on this topic. Three courses are in the 'Programming' cluster, indicating design and programming are sometimes taught together. Word cloud analysis highlighted 'software', 'design', 'model', 'system', 'architecture', 'analysis', 'engineering', and 'requirement'.

Construction & Programming: Most numerous KA with 75 courses across all universities. Undergraduate courses dominate (62 of 75). Only 12 courses are in SE-specific tracks. Within 75 courses, 55 are in the 'programming' cluster. MSc-level courses cover advanced topics including parallel programming, web and cloud computing, cryptography and security, and functional programming.

Testing: Nineteen courses across nine universities. Most courses at MSc level (12 of 19). Eleven courses are in SE-specific tracks. Signature courses are the majority (12 of 19), most at MSc level (9 of 12). Word cloud analysis revealed keywords including 'testing', 'test', 'software', 'system', 'technique', 'model', 'code', 'program', 'quality', and 'risk'.

Operations: Fifteen courses across all universities. More recurrent at MSc level (9 of 15) but also present at BSc level (6 of 15). Seven courses in SE tracks. Seldom considered as signature courses (2 of 15), exceptions being Software Containerization at VU and DevOps and Cloud-Based Systems at UvA.

Maintenance: Nineteen courses across seven universities. Almost all at MSc level (18 of 19). Majority in SE-specific programs (15 of 19). Signature courses tightly coupled with software evolution. Word cloud analysis highlighted 'software', 'software evolution', 'software system', 'quality', 'metric', 'techniques', 'code', and 'tools'.

Configuration Management: Infrequent KA with only 9 courses across six universities. Balanced between BSc (4) and MSc (5). Inexistence of signature courses. One seminar, two programming cluster courses, five in SE-specific tracks.

SE Management: Popular KA with 24 courses across all but one university. Only 6 BSc courses. Five signature courses identified. Eleven courses in SE-specific tracks. Word cloud analysis revealed 'software', 'management', 'process', 'development', and 'model'.

SE Process: Popular KA with 25 courses, all universities providing at least two courses. Mostly MSc curricula (17 of 25). Only one signature course. Fifteen courses in SE-specific tracks.

SE Models: Popular KA with 28 courses evenly split between BSc and MSc. Nine universities offer courses. Five signature courses identified. Tight interplay with Design KA. Word cloud analysis highlighted 'system', 'model', 'software', 'engineering', 'development', 'tool', 'design', 'information', 'model-driven', 'language', and 'transformation'.

Security: Sixteen courses across nine universities. More popular at MSc level (11 courses). Five signature courses. Two seminars offered. Word cloud analysis revealed 'system', 'analysis', 'vulnerability', 'secure', 'programming', 'software development', and 'programming language'.

SE Economics: Nine courses across seven universities. Two BSc, seven MSc. Two MSc seminars. Six in SE-specific tracks. Word cloud analysis highlighted 'software', 'product', 'management', 'software product', 'software ecosystem', and 'business'.

Verification: Twenty-eight courses across all universities. Mostly MSc (18 courses), with ten BSc courses. Eight universities include Verification at BSc level, all ten at MSc level. Almost half (14) in SE-specific tracks. Three seminars. Word cloud analysis revealed 'program', 'system', 'technique', 'software', 'security', and 'model checking'.

PL Design: Twenty-four courses, half at BSc level, half at MSc level. BSc courses by nine universities, MSc by six universities. Nine courses in SE-specific tracks. Six compiler construction courses, four programming language concepts courses, two domain-specific language courses. Word cloud analysis highlighted 'language', 'programming language', 'program', 'type', 'semantic', 'compiler', 'concept', 'programming', 'technique', 'imperative', 'security', and 'system'.

6.3 Correlations Among Knowledge Areas (RQ2)

Spearman correlation analysis revealed three clusters of tightly coupled knowledge areas:

Cluster 1: Requirements, Architecture, Design, SE Models

Requirements shows weak correlations with Design (ρ=0.34), SE Models (ρ=0.30), and Architecture (ρ=0.25). Architecture shows weak correlations with Design (ρ=0.31) and SE Models (ρ=0.27). Design shows moderate correlation with SE Models (ρ=0.42). SE Models shows moderate correlation with Design (ρ=0.43) and weak correlations with Requirements (ρ=0.30) and Architecture (ρ=0.27). This cluster reflects the typical positioning of these KAs as phases following requirements engineering in software development.

Cluster 2: Testing, Security, Verification

This cluster shows looser correlations. Testing has weak correlations with Verification (ρ=0.21) and Software Security (ρ=0.15). Software Security has weak correlation with Verification (ρ=0.21). Despite being linked conceptually, these topics are often addressed in silos.

Cluster 3: Operations, SE Management, SE Process, Maintenance, Configuration Management

This cluster shows the highest cohesion. Moderate correlations exist between SE Process and Operations (ρ=0.41), SE Process and SE Management (ρ=0.42), and SE Management and Operations (ρ=0.42). Topics related to DevOps are seldom taught singularly, suggesting either theoretical concepts are interdependent, topics lack depth individually, or SWEBOK dissects higher concepts into finer-grained KAs.

Lone Riders: Construction & Programming, PL Design, and SE Economics appear isolated. Construction & Programming is foundational and covered independently. PL Design is theoretical and formal. SE Economics shows weak correlations with SE Management (ρ=0.21) and SE Process (ρ=0.13).

6.4 University Educational Foci (RQ3)

Analysis of KA distribution across universities reveals more commonalities than discrepancies. Most universities provide high numbers of Construction & Programming courses, reflecting the foundational nature of programming, especially prevalent at BSc level. UvA shows the highest recurrence due to Introduction to Programming courses duplicated across curricula.

Less recurrent KAs such as Architecture, Verification, and SE Economics show overall low recurrence across all universities, with UT exception for verification specialization. Software Security varies from limited fraction (RU) to notable portion (TUD). PL Design is notable at RU but less recurrent elsewhere.

No institution delivers a narrow and specialized focus on a specific SE topic. The vast range of topics covered suggests the multifaceted skills required to train modern software engineers are necessary regardless of specific SE specializations.

7. Synthesis and Discussion

7.1 Integrating Predictive Analytics, Resilient Design, and Education

This research demonstrates the interconnected nature of software engineering practice, system resilience, and professional education. The high accuracy achieved by the Random Forest bug prediction framework (99.98%) and DistilBERT model (99.39%) shows that machine learning can effectively address the challenge of early defect identification. These models enable developers to allocate resources efficiently, prioritize testing efforts, and produce more resilient software solutions. The SHAP interpretability analysis provides transparency by identifying the most influential factors in defect prediction, supporting informed decision-making in software quality assurance.

The fault tolerance analysis reveals that advanced software modeling techniques provide systematic approaches to

designing resilient distributed systems. State machines, actor-based models, and Petri nets offer complementary strengths for representing, analyzing, and validating fault-tolerant architectures. The integration of predictive modeling with proactive recovery mechanisms enables systems to anticipate and mitigate faults before they impact operations. Simulation-based approaches provide controlled environments for testing and refining fault management strategies.

The educational analysis identifies both strengths and gaps in software engineering curricula across Dutch universities. While Construction & Programming is universally well-covered, reflecting its foundational importance, other KAs show varied coverage. The clusters of tightly coupled KAs suggest opportunities for integrated teaching approaches. The identified gaps in SE Economics and Configuration Management coverage highlight areas for curriculum development.

7.2 Implications for Practice

For software practitioners, the bug prediction framework offers a validated approach for improving software quality through early defect identification. The Random Forest model's exceptional performance and interpretability make it suitable for real-world deployment. The feature engineering approach, incorporating complexity_score, churn_per_commit, bug_fix_ratio, dev_productivity, risk_index, and comment_bucket, provides a comprehensive set of predictors that can be applied across diverse software projects.

For system architects, the fault tolerance analysis offers guidance on selecting appropriate modeling techniques for specific system characteristics. State machines are suitable for well-defined state spaces with clear transition patterns. Actor models excel in environments requiring strong fault isolation and asynchronous communication. Petri nets provide comprehensive capabilities for analyzing concurrency, resource sharing, and fault propagation. The integrated framework combining predictive modeling, proactive recovery, and simulation-based approaches offers a comprehensive strategy for resilient system design.

7.3 Implications for Education

For software engineering educators, the curriculum analysis provides empirical evidence for curriculum development and refinement. The clusters of tightly coupled KAs suggest opportunities for integrated teaching approaches that help students understand relationships between knowledge areas. The underrepresentation of SE Economics and Configuration Management indicates areas for potential curriculum enhancement.

The finding that many KAs are primarily taught at MSc level suggests potential for earlier introduction of advanced topics to better prepare students for professional practice. The diversity of course types (programming, PL design, seminar, SE101, project) across institutions demonstrates varied pedagogical approaches that can inform course design decisions.

The substantial interrater agreement in KA annotation (Cohen's kappa = 0.718) validates the mapping methodology and suggests its potential applicability in other educational contexts. Researchers are invited to apply the methodology in their own geographical regions to contrast software engineering education programs globally.

7.4 Balancing Predictive Accuracy and Computational Efficiency

The comparison between Random Forest and DistilBERT highlights the trade-off between predictive accuracy and computational efficiency. Random Forest achieved marginally higher accuracy (99.98% vs. 99.39%) with lower training time (110.87s vs. 133.8s), making it more computationally efficient. However, DistilBERT's deep learning approach offers advantages in handling unstructured data and capturing complex contextual patterns.

For practitioners selecting bug prediction approaches, this trade-off should be considered in context. Random Forest may be preferred for its computational efficiency and interpretability, particularly in resource-constrained environments or when model transparency is important. DistilBERT may be preferred for its deep learning capabilities and potential to capture more nuanced patterns, particularly in complex software environments with abundant textual data.

7.5 Scalability and Complexity in Fault Tolerance

The fault tolerance analysis reveals ongoing challenges in balancing scalability with fault management efficiency. Consensus algorithms like Paxos and Raft become impractical as system size increases due to extensive inter-node communication. Redundancy-based approaches lead to resource wastage in larger environments due to multiple backups or replicas.

Emerging approaches including hierarchical fault management and partitioning offer promising solutions by grouping nodes into smaller fault domains to reduce communication overhead and isolate failures more effectively. Predictive analytics and machine learning optimize fault management processes by anticipating faults and deploying resources dynamically. The integration of these approaches will be critical for ensuring fault tolerance in next-generation distributed systems.

7.6 Curriculum Alignment and Industry Needs

The educational analysis reveals significant alignment between Dutch university curricula and established knowledge areas, while also highlighting gaps. The limited coverage of SE Economics and Configuration Management may reflect historical curriculum development rather than diminished importance. These areas are highly relevant to industry practice and should be considered for curriculum enhancement.

The prevalence of Construction & Programming across all universities reflects the foundational nature of programming skills, consistent with findings in related studies (Murphy et al., 2016; Simon et al., 2018). The balance between BSc and MSc coverage varies across KAs, with some areas (Maintenance, Operations, Verification) predominantly at MSc level. This distribution may reflect assumptions about the appropriate

educational level for different topics, which could be revisited to ensure comprehensive coverage across both undergraduate and graduate programs.

8. Limitations and Future Work

8.1 Bug Prediction Limitations

The bug prediction study has several limitations. The models were trained and tested on a single dataset (Kaggle software defect prediction dataset), which may limit generalizability to other software projects or real-world environments. While the dataset includes diverse software quality metrics, it may not capture all relevant factors influencing defect likelihood. The SMOTE oversampling technique, while effective for addressing class imbalance, creates synthetic samples that may not fully represent real-world defect characteristics.

Future work should validate the models on larger and more diverse datasets from multiple software projects, including those with different programming languages, development processes, and application domains. Cross-project prediction experiments would assess model generalizability across different software contexts. Real-time deployment in CI/CD environments would evaluate practical utility and identify integration challenges.

8.2 Fault Tolerance Limitations

The fault tolerance analysis is based on existing literature and does not include empirical evaluation of specific modeling techniques in controlled experiments. The scalability of state machines, Petri nets, and actor models in very large distributed systems requires further investigation. The integration of predictive analytics with proactive recovery mechanisms presents implementation challenges that warrant detailed case studies.

Future research should conduct empirical evaluations of modeling techniques in real-world distributed systems, comparing effectiveness across different application domains and fault scenarios. The development of lightweight fault-tolerant models suitable for resource-constrained environments (edge computing, IoT) should be prioritized. Advanced AI approaches including reinforcement learning and self-healing systems offer promising directions for automating fault detection, diagnosis, and recovery.

8.3 Educational Analysis Limitations

The educational analysis is limited to Dutch research universities and 207 courses, which may not represent software engineering education in other countries or institutional types. The analysis relies on course syllabi rather than direct observation of teaching practice, and syllabi content may deviate from actual course content. The KA annotation process, while rigorous with substantial interrater agreement, involves subjective interpretation of course content.

Future research should extend the analysis to other countries and institutional types to enable international comparison. Direct observation of teaching practice and analysis of student outcomes would provide richer understanding of educational effectiveness. Longitudinal studies tracking curriculum evolution would reveal trends and identify emerging topics requiring attention.

9. Conclusion

This research has presented a comprehensive investigation spanning three critical dimensions of software engineering: bug prediction using machine learning frameworks, fault tolerance modeling for distributed systems, and systematic analysis of software engineering education. Through empirical evaluation, we demonstrated that Random Forest models achieve exceptional performance in software defect prediction with 99.98% accuracy, outperforming baseline models including ANN, CNN, Decision Tree, AdaBoost, and SVM. The SHAP-based interpretability analysis identified past defects, static analysis warnings, cyclomatic complexity, test coverage, and risk index as the most influential factors in defect prediction, providing transparency and supporting informed decision-making.

Our analysis of fault tolerance mechanisms revealed that advanced software modeling techniques—including state machines, Petri nets, and actor-based frameworks—provide systematic approaches for designing resilient distributed systems capable of maintaining operational continuity despite component failures. The integration of predictive modeling, proactive recovery mechanisms, and simulation-based approaches offers a comprehensive framework for fault management. The actor model provides strong fault isolation and asynchronous communication, Petri nets excel at modeling concurrency and resource dependencies, and state machines offer systematic representation of system states and transitions.

The systematic examination of 207 courses across 10 Dutch universities identified critical knowledge area coverage patterns and correlations. Three clusters of tightly coupled knowledge areas emerged: (i) requirements, architecture, and design; (ii) testing, verification, and security; and (iii) process-oriented and DevOps topics. The analysis highlighted underrepresentation of software engineering economics and configuration management topics, suggesting opportunities for curriculum enhancement. The substantial interrater agreement validates the mapping methodology and suggests its potential applicability in other educational contexts.

The integration of these three perspectives offers a holistic framework for advancing software engineering practice through predictive analytics, resilient system design, and evidence-based curriculum development. Practitioners can deploy the bug prediction framework for early defect identification, system architects can leverage advanced modeling techniques for fault-tolerant system design, and educators can use the curriculum analysis for evidence-based curriculum development and refinement.

Future work should focus on validating the bug prediction models on larger and more diverse datasets, conducting empirical evaluations of fault tolerance techniques in real-world distributed systems, and extending the educational analysis to other countries and institutional types. The integration of emerging technologies—including quantum computing, blockchain, and AI-driven systems—presents

opportunities for transformative advances in bug prediction, fault tolerance, and software engineering education. By addressing these opportunities, researchers and practitioners can continue to advance the field and build more reliable, resilient, and effective software systems.

References

Al-Sulbi, K., & Attaallah, A. (2025). Symmetric bug prediction in software requirement by machine learning algorithms. *Scientific Reports, 15*(1), 38276. https://doi.org/10.1038/s41598-025-22193-x

Ali, A., Xia, Y., Umer, Q., & Osman, M. (2024). BERT based severity prediction of bug reports for the maintenance of mobile applications. *Journal of Systems and Software*, 111898. https://doi.org/10.1016/j.jss.2023.111898

Alsaedi, S. A., Noaman, A. Y., Gad-Elrab, A. A. A., & Eassa, F. E. (2023). Nature-based prediction model of bug reports based on ensemble machine learning model. *IEEE Access*. https://doi.org/10.1109/ACCESS.2023.3288156

Arasteh, B., Sefati, S. S., Popovici, E. C., Ince, I. F., & Kiani, F. (2025). A bedbug optimization-based machine learning framework for software fault prediction. *Mathematics*. https://doi.org/10.3390/math13213531

Avizienis, A. (1978). Fault-tolerance: The survival attribute of digital systems. *Proceedings of the IEEE, 66*(10), 1109-1125.

Avizienis, A., & Kelly, J. P. (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer, 17*(08), 67-80.

Bharath, K. S., & Jagadeesh, P. (2023). An innovative software bug prediction system using random forest algorithm for enhanced accuracy in comparison with logistic regression algorithm. In *2023 Intelligent Computing and Control for Engineering and Business Systems (ICCEBS)*.

Bondavalli, A., Chiaradonna, S., Di Giandomenico, F., & Xu, J. (2002). An adaptive approach to achieving hardware and software fault tolerance in a distributed computing environment. *Journal of Systems Architecture, 47*(9), 763-781.

Dao, A. H., & Yang, C. Z. (2021). Severity prediction for bug reports using multi-aspect features: A deep learning approach. *Mathematics*. https://doi.org/10.3390/math9141644

Dobre, C., Pop, F., & Cristea, V. (2011). New trends in large scale distributed systems simulation. *Journal of Algorithms & Computational Technology, 5*(2), 221-257.

Du, X. et al. (2022). CoreBug: Improving effort-aware bug prediction in software systems using generalized k-core decomposition in class dependency networks. *Axioms, 11*(5). https://doi.org/10.3390/axioms11050205

Fan, G., Liang, Y., Zu, L., Yu, H., Huang, Z., & Chen, W. (2026). Automatic identification of extrinsic bug reports for just-in-time bug prediction. *Science of Computer Programming, 249*, 103410. https://doi.org/10.1016/j.scico.2025.103410

Gao, Z., Cecati, C., & Ding, S. X. (2015). A survey of fault diagnosis and fault-tolerant techniques—Part I: Fault diagnosis with model-based and signal-based approaches. *IEEE Transactions on Industrial Electronics, 62*(6), 3757-3767.

Gartner, F. C. (1999). Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR), 31*(1), 1-26.

Hanmer, R. S. (2013). *Patterns for fault tolerant software*. John Wiley & Sons.

Heeren, B., Dalpiaz, F., Seraj, M., Verdecchia, R., & Zaytsev, V. (2026). A systematic analysis of higher education on software engineering in the Netherlands. *Journal of Systems and Software*.

Holzmann, G. J. (2004). *The SPIN model checker: Primer and reference manual*. Addison-Wesley.

Hou, Z., Gong, L., Yang, M., Zhang, Y., & Yang, S. (2022). Software bug prediction based on complex network considering control flow. In *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)* (pp. 246-254). IEEE. https://doi.org/10.1109/QRS-C57518.2022.00044

Huang, X., Zhang, H., Zhou, X., Shao, D., & Jaccheri, L. (2021). A research landscape of software engineering education. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)* (pp. 181-191). IEEE.

Jasz, J. (2024). The effectiveness of hidden dependence metrics in bug prediction. *IEEE Access*. https://doi.org/10.1109/ACCESS.2024.3406929

Jumare, H., Darius, & Chinyio, T. (2024). Software defect prediction using machine learning and deep learning techniques. *Kasu Journal of Computer Science, 1*(3), 527-543. https://doi.org/10.47514/kjcs/2024.1.3.0010

Kabir, S. M. H., Rahman, M. T., & Mridul, A. H. (2025). Software defect prediction using traditional machine learning and ensemble learning algorithms. *Smart Wearable Technology*. https://doi.org/10.47852/bonviewswt52025645

Kesavan, E. (2025). Software bug prediction using machine learning algorithms: An empirical study on code quality and reliability. *International Journal of Innovative Science, Engineering and Management*, 377-381. https://doi.org/10.69968/ijisem.2025v4i3377-381

Khleel, N. A. A., & Nehez, K. (2023). A novel approach for software defect prediction using CNN and GRU based on SMOTE Tomek method. *Journal of Intelligent Information Systems*. https://doi.org/10.1007/s10844-023-00793-1

Koren, I., & Krishna, C. M. (2020). *Fault-tolerant systems*. Morgan Kaufmann.

Kumari, M., Singh, R., & Singh, V. B. (2025). Prioritization of software bugs using entropy-based measures. *Journal of Software: Evolution and Process, 37*(2). https://doi.org/10.1002/smr.2742

Kumari, P., & Kaur, P. (2021). A survey of fault tolerance in cloud computing. *Journal of King Saud University - Computer and Information Sciences, 33*(10), 1159-1176.

Lee, J., & Cheng, Y. C. (2011). Change the face of software engineering education: A field report from Taiwan. *Information and Software Technology, 53*(1), 51-57.

Liargkavas, G., Papadopoulou, A., Kotti, Z., & Spinellis, D. (2021). Software engineering education knowledge. In *Proceedings of the 2021 ACM/IEEE International Conference on Software Engineering*.

Malik, B., & Zafar, S. (2012). A systematic mapping study on software engineering education. *International Journal of Educational and Pedagogical Sciences, 6*(11), 3343-3353.

Mansour, I., Said, M. B., & Kacem, Y. H. (2025). Enhanced software bug prediction using double-stacked ensembles and halving search optimization. *Procedia Computer Science, 270*, 3789-3798. https://doi.org/10.1016/j.procs.2025.09.504

Murphy, E., Crick, T., & Davenport, J. H. (2016). An analysis of introductory programming courses at UK universities. *arXiv preprint arXiv:1609.06622*.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., & Deardeuff, M. (2015). How Amazon Web Services uses formal methods. *Communications of the ACM, 58*(4), 66-73.

Pradel, M., & Sen, K. (2018). DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*. https://doi.org/10.1145/3276517

Qadir, M. M., & Usman, M. (2011). Software engineering curriculum: A systematic mapping study. In *2011 Malaysian Conference in Software Engineering* (pp. 269-274). IEEE.

Rehman, A. U., Aguiar, R. L., & Barraca, J. P. (2022). Fault-tolerance in the scope of cloud computing. *IEEE Access, 10*, 63422-63441.

Simon, Mason, R., Crick, T., Davenport, J. H., & Murphy, E. (2018). Language choice in introductory programming courses at Australasian and UK universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 852-857). ACM.

Slatten, V., Herrmann, P., & Kraemer, F. A. (2013). Model-driven engineering of reliable fault-tolerant systems—A state-of-the-art survey. *Advances in Computers, 91*, 119-205.

Spichkova, M., Thomas, I. E., Schmidt, H. W., Yusuf, I. I., Drumm, D. W., Androulakis, S., ... & Russo, S. P. (2015). Scalable and fault-tolerant cloud computations: Modelling and implementation. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)* (pp. 396-404). IEEE.

Tenhunen, S., Mannistö, T., Luukkainen, M., & Ihantola, P. (2023). A systematic literature review of capstone courses in software engineering. *Information and Software Technology*, 107191.

Xu, B. et al. (2025). Cross-project aging-related bug prediction based on transfer learning and class imbalance learning. *IEEE Transactions on Dependable and Secure Computing*. https://doi.org/10.1109/TDSC.2025.3567957