

Automated Detection of Software Defects Through Machine Learning and Source Code Analysis

Dr. Daniel Lim Wei Jian

Assistant Professor, Penang Institute of Code Intelligence, Malaysia

Abstract

Software defects represent a persistent challenge in software engineering because incorrect program behavior can affect system reliability, maintainability, operational efficiency, and user confidence. Conventional software quality assurance relies on techniques such as code inspection, static analysis, and software testing to identify potential problems before deployment. Although these approaches provide essential mechanisms for evaluating software quality, large and frequently changing codebases can make comprehensive inspection resource-intensive. Machine learning introduces opportunities to analyze historical defect information and source code characteristics to identify components that may require additional verification.

The present paper develops a hypothetical methodological framework for examining automated software defect detection through source code analysis and machine learning. The investigation focuses on predicting defect-prone source files rather than claiming to identify the exact location of every programming error. Four approaches are considered: rule-based static analysis, Logistic Regression, Random Forest, and Gradient Boosting. A fictional dataset containing 12,000 source files distributed across 30 software repositories is introduced to demonstrate the proposed research design. Static code complexity, coupling, code churn, and analyzer-generated warning counts are considered principal explanatory characteristics.

The proposed methodology integrates source code parsing, metric extraction, historical defect labeling, feature preprocessing, project-independent validation, and comparative classification assessment. Precision, recall, and F1-score are selected as principal performance measures. Two illustrative analytical tables present the hypothetical defect-label distribution and comparative classification outcomes. All numerical values are manually constructed and do not represent actual source code analysis, trained machine learning models, or completed experiments.

The methodological discussion examines defect-label reliability, class imbalance, project-specific variation, temporal information leakage, false-positive predictions, and model interpretability. Particular attention is given to distinguishing predicted defect risk from confirmed software defects. The paper concludes that combining machine learning with source code analysis provides a relevant methodological approach for prioritizing software quality assurance activities, although genuine effectiveness requires reliable defect records, reproducible experimentation, independent validation, and developer-centered assessment.

Keywords: Software Defect Detection, Machine Learning, Source Code Analysis, Software Quality Assurance, Static Code Analysis, Defect Prediction, Random Forest, Gradient Boosting, Code Complexity, Software Reliability.

1. Introduction

Software systems play an increasingly important role in contemporary information processing, organizational operations, digital communication, and technological infrastructure. The reliability of these systems depends substantially on the quality of their source code and the effectiveness of procedures used to identify potential programming defects. Software defects can arise from incorrect logic, inadequate handling of exceptional conditions, inappropriate interactions among components, and unintended consequences of software modifications. Identifying such problems before deployment is therefore an important objective of software quality assurance. However, the increasing size and complexity of modern applications create difficulties for development teams attempting to inspect every source code component through conventional procedures.

Traditional software defect detection involves activities such as code review, static program analysis, unit testing, integration testing, and debugging. These procedures examine software behavior or structural characteristics through different analytical mechanisms. Static analysis evaluates source code without executing the application and can identify selected rule violations, suspicious constructs, and potential programming problems. Dynamic testing examines software behavior under specified execution conditions. Although these approaches are complementary, neither guarantees the identification of every defect. Their effectiveness depends on test coverage, analysis rules, code characteristics, and the resources available for software inspection.

Machine learning provides an additional approach for examining software quality by identifying statistical relationships between source code characteristics and historical defect records. Predictive models can use information such as lines of code, cyclomatic complexity, coupling, code churn, and static analyzer warnings to estimate the likelihood that a software component belongs to a defect-prone category. These estimates can help development teams prioritize code inspection and testing activities. However, a classifier assigning high defect risk to a source file does not establish that the file contains a particular programming error. Predictive classification and confirmed defect detection must therefore remain clearly distinguished.

Software defect prediction introduces methodological challenges associated with incomplete issue records, inconsistent defect labels, changing source code structures, and differences among software projects. Historical defect information is frequently derived from issue tracking systems and version-control records, which may contain incomplete or incorrectly categorized reports. Furthermore, code characteristics associated with defects in one project may provide limited predictive information in another. Classification performance can also be distorted when related files, duplicated code, or future development information appear across training and testing datasets. These limitations demonstrate the importance of transparent data preparation and independent evaluation.

The present paper proposes a hypothetical methodological framework for evaluating automated software defect detection through source code analysis and machine learning. Four alternative configurations are examined using a fictional collection of Java source files from independent repositories. A rule-based static analysis baseline is compared with Logistic Regression, Random Forest,

and Gradient Boosting. The proposed investigation emphasizes source code metrics, historical defect identification, project-independent testing, and minority-class evaluation. Two illustrative tables demonstrate dataset composition and comparative classification performance. No actual source code repositories were analyzed, and no machine learning experiments were conducted. The paper establishes a structured approach for future empirical investigation.

2. Objectives of the Study

The primary objective of this study is to examine the methodological application of machine learning and source code analysis for identifying software components associated with elevated defect risk. The investigation focuses on code characteristics, classification performance, and predictive reliability. Three specific objectives establish the analytical direction of the proposed research framework.

2.1 Examination of Source Code Characteristics Associated With Defects

The first objective is to investigate how structural and historical characteristics of source code can be represented for automated defect prediction. Software components differ in size, control-flow complexity, coupling, modification frequency, and the number of warnings generated by static analysis tools. These characteristics may provide useful information concerning the likelihood of defect occurrence. The proposed study considers how source-level metrics and development-history indicators can be extracted consistently from software repositories and incorporated into predictive classification models.

The investigation further aims to distinguish potentially informative software metrics from characteristics that merely correlate with unrelated project conditions. Large source files may contain more defects because they include more code, but size alone cannot establish poor software quality. Similarly, frequent modifications may reflect corrective maintenance, active feature development, or changing requirements. Consequently, the proposed framework emphasizes combined feature assessment and careful interpretation. This objective seeks to identify methodological requirements for analyzing source code without treating individual complexity metrics as definitive evidence of defects.

2.2 Comparative Evaluation of Machine Learning Defect Prediction Models

The second objective focuses on establishing a comparative framework for evaluating rule-based analysis and machine learning classification. Logistic Regression, Random Forest, and Gradient Boosting are selected to represent different statistical and computational approaches. A rule-based static analysis configuration provides a practical baseline for comparison. The intended evaluation examines the ability of each method to identify source files subsequently associated with confirmed defect reports. Precision, recall, and F1-score are considered the principal performance measures.

A further objective involves examining whether nonlinear machine learning methods provide useful predictive advantages over simpler classification procedures. Random Forest and Gradient Boosting can represent interactions among source code complexity, coupling, and historical modification characteristics. However, additional model complexity may introduce greater computational requirements and sensitivity to project-specific patterns. Consequently, the proposed framework emphasizes fair comparison using consistent input information, independent testing, and

appropriate model selection. Algorithmic sophistication is not assumed to guarantee superior defect prediction.

2.3 Assessment of Defect Prediction Reliability and Practical Applicability

The third objective is to identify methodological factors that may influence the reliability of automated defect prediction systems. Incorrect defect labels, duplicated source files, imbalanced classification categories, and differences among software repositories can affect model performance. The proposed investigation considers how verified issue reports, project-independent partitioning, and appropriate evaluation measures could reduce selected difficulties. Particular attention is given to preventing source code information from the test repositories from influencing training procedures.

The investigation also considers the practical requirements associated with using predictive defect information during software development. Development teams must interpret model predictions alongside static analyzer findings, testing outcomes, and the functional importance of software components. False-positive predictions may increase unnecessary inspection effort, whereas false negatives can leave important files without additional review. Consequently, this objective emphasizes interpretability, transparent uncertainty assessment, and responsible integration with established software quality assurance procedures.

3. Review of Literature

3.1 Early Research on Source Code Metrics and Defect Prediction

Nagappan and Ball (2005) investigated relative code churn measures and examined their ability to predict software defect density. Nagappan et al. (2006) subsequently studied software component failures using source code metrics and demonstrated relationships between selected complexity characteristics and post-release defects. Menzies et al. (2007) examined the use of static code attributes for learning defect predictors. These contributions established important foundations for analyzing software quality through measurable code characteristics and demonstrated that predictive performance depends on how extracted metrics are used within learning procedures.

Kim et al. (2008) investigated the classification of software changes as clean or buggy using historical repository information. Moser et al. (2008) compared change metrics with static source code attributes for defect prediction. Their findings emphasized that process-related information can complement structural code metrics. Lessmann et al. (2008) developed a comparative framework for benchmarking defect prediction classifiers and examined limitations of inconsistent evaluation procedures. Together, these studies established the importance of combining appropriate software metrics with rigorous model comparison.

3.2 Software Repository Mining and Defect Label Reliability

Bird et al. (2009) examined bias in datasets constructed from software bug-fix histories and identified limitations in the completeness of recorded defect information. Hassan (2009) investigated the complexity of code changes as a basis for predicting software faults. Turhan et al. (2009) examined the transferability of defect prediction models across organizational datasets. These contributions demonstrated that software defect prediction depends on both source code characteristics and the quality

of historical development information. They also highlighted challenges associated with transferring models between different projects.

D'Ambros et al. (2010) compared alternative bug prediction approaches and demonstrated the importance of reproducible benchmarking. Hall et al. (2012) conducted a systematic literature review of software fault prediction research and examined the influence of modeling techniques, software metrics, and research context. Herzig et al. (2013) investigated the consequences of incorrectly categorized issue reports for defect prediction datasets. Together, these studies emphasized that trustworthy defect prediction requires validated labels, transparent data construction, and careful interpretation of reported predictive results.

3.3 Machine Learning Evaluation and Software Quality Prediction

Breiman (2001) introduced Random Forest as an ensemble classification method capable of modeling nonlinear relationships. Friedman (2001) developed Gradient Boosting through sequential predictive optimization. He and Garcia (2009) examined learning difficulties associated with imbalanced classification data. Cawley and Talbot (2010) investigated model-selection bias, while Saito and Rehmsmeier (2015) emphasized the importance of precision–recall evaluation when positive observations are less frequent. These contributions established methodological principles directly relevant to software defect prediction.

Kamei et al. (2013) examined just-in-time quality assurance through the prediction of risky software changes. Rahman and Devanbu (2013) investigated the relative effectiveness of process metrics and code metrics for defect prediction. Shepperd et al. (2014) examined researcher bias in machine learning-based software defect prediction studies. These publications indicate that reliable predictive performance depends on dataset construction, model development, and evaluation procedures. Their contributions support comparisons that account for project differences and avoid assuming that one algorithm is universally preferable.

4. Research Methodology

4.1 Research Design and Source Code Analysis Framework

The present paper adopts a hypothetical comparative quantitative research design for investigating automated software defect prediction. Four configurations are proposed: rule-based static analysis, Logistic Regression, Random Forest, and Gradient Boosting. The unit of analysis is an individual source file recorded at a defined pre-release snapshot. The intended classification target indicates whether that file is subsequently associated with at least one validated post-release defect during a 90-day observation period. This operational definition supports file-level risk prediction rather than exact localization of programming errors.

The proposed analytical workflow consists of repository selection, source code parsing, static metric extraction, historical feature construction, defect-label verification, model development, and independent evaluation. Abstract syntax tree parsing would provide structural information about source files. Version-control histories would support calculation of development activity metrics. Static analyzer warnings would provide additional predictors. Machine learning models would use the resulting structured feature records to estimate defect likelihood. No repositories were inspected, and no classifiers were trained for this illustrative manuscript.

4.2 Hypothetical Dataset and Software Metric Extraction

For illustrative purposes, the proposed study assumes a fictional collection of 30 independent Java software repositories. Each repository contributes 400 source files observed at a predefined release snapshot, producing 12,000 file-level records. The dataset contains 9,600 files with no linked validated post-release defect and 2,400 files with at least one qualifying defect report. Each repository is assumed to contain 320 negative and 80 positive observations. These values are constructed examples and do not represent actual software projects or verified defect frequencies.

The proposed source code features would include logical lines of code, cyclomatic complexity, nesting depth, method count, coupling indicators, and static warning counts. Historical process features would include file modification frequency and lines added or deleted during the 30 days preceding the release snapshot. Defect labels would be established by reviewing validated issue reports and linking corrective changes to the appropriate pre-release files. Feature extraction would exclude modifications occurring after the prediction snapshot, while repository identifiers would be retained for partitioning and excluded from the predictive feature set.

4.3 Model Development and Comparative Evaluation

The proposed repository-level partition would allocate 18 projects to training, six projects to validation, and six projects to independent testing. This division produces 7,200 training observations, 2,400 validation observations, and 2,400 test observations. Rule-based static analysis would classify files according to predefined complexity, churn, and warning thresholds optimized using development data. Logistic Regression would provide a statistical baseline, while Random Forest and Gradient Boosting would estimate nonlinear relationships. Any feature scaling or preprocessing would be learned exclusively from the appropriate development partitions.

The principal evaluation would report precision, recall, and F1-score for the defect-linked category. Precision would describe the proportion of files flagged as defect-prone that have qualifying defect labels, while recall would describe the proportion of labeled defect-prone files successfully identified. F1-score would summarize their balance. A completed study should additionally report precision–recall area, probability calibration, project-level performance, and inspection effort. Statistical uncertainty would require repository-aware resampling because files within the same project may not represent independent observations.

5. Analysis

5.1 Illustrative Distribution of Defect-Prone Source Files

Software defect prediction datasets frequently contain unequal numbers of source files with and without recorded defects. This distribution can influence classifier development and produce misleading performance interpretations when aggregate accuracy is considered without minority-class recognition.

Table 1 presents the hypothetical distribution of defect-linked and non-defect-linked source files across repository-independent evaluation partitions. The dataset contains 80% negative observations and 20% positive observations. These proportions are manually specified to demonstrate the intended evaluation design.

The illustrative distribution also emphasizes the importance of separating repositories during training and testing. Source files from the same project can share programming conventions,

dependencies, architectural characteristics, and development practices. Randomly distributing these files across evaluation subsets may produce favorable estimates that do not generalize to unfamiliar repositories. The proposed framework therefore maintains project-level independence. However, such separation does not guarantee generalization across different programming languages, development environments, or software application categories.

Table 1: Hypothetical Distribution of Software Source Files and Defect Labels

Source File Category	Training Files	Validation Files	Test Files	Total
No Linked Post-Release Defect	5,760	1,920	1,920	9,600
Linked Validated Post-Release Defect	1,440	480	480	2,400
Total	7,200	2,400	2,400	12,000

The fictional test partition contains 480 defect-linked files and 1,920 files without qualifying defect links. These observations provide the assumed reference distribution for the comparative classification example.

5.2 Illustrative Comparison of Automated Defect Prediction Models

Automated defect prediction systems can be compared by examining their ability to prioritize source files associated with validated defects. Precision measures the reliability of positive classifications, while recall indicates the proportion of defect-linked files successfully identified. Table 2 presents hypothetical classification outcomes for four methods evaluated on the same fictional test partition. The reported counts were manually selected and do not represent actual static analyzer outputs or machine learning predictions.

The illustrative comparison assigns the highest F1-score to Gradient Boosting, followed by Random Forest, Logistic Regression, and the rule-based static analysis baseline. This constructed pattern demonstrates how different analytical approaches could be compared under consistent evaluation conditions. However, the assigned values cannot establish experimental superiority. A genuine investigation would require implemented analysis tools, verified defect labels, independent test predictions, and appropriate statistical comparisons before identifying a preferred detection strategy.

Table 2: Hypothetical Classification Performance of Software Defect Prediction Approaches

Defect Prediction Approach	True Positives	False Positives	Precision	Recall	F1-Score
Rule-Based Static Analysis	216	240	0.474	0.450	0.462
Logistic Regression	264	180	0.595	0.550	0.571
Random Forest	312	156	0.667	0.650	0.658
Gradient Boosting	336	144	0.700	0.700	0.700

The illustrative results demonstrate potential trade-offs between defect identification and unnecessary inspection. The corresponding accuracy values would be 79.0%, 83.5%, 86.5%, and 88.0%, respectively. However, accuracy alone would be insufficient because a classifier predicting every file as non-defective would achieve 80% accuracy while identifying no defect-linked files.

6. Discussion

6.1 Source Code Complexity and Historical Defect Indicators

Source code complexity and development history can provide useful information for identifying potentially defect-prone software components. Files containing complicated control flow, extensive dependencies, or frequent modifications may present challenges during maintenance and testing. Historical development indicators can describe how source files have changed over time, while static metrics provide information concerning their structural characteristics at a particular snapshot. However, these characteristics do not independently establish the existence of programming defects. Their usefulness depends on the software context and the consistency of the available defect labels.

The proposed framework combines code-level and process-related characteristics rather than treating one metric as sufficient for classification. This approach reflects the methodological findings of Moser et al. (2008), Hassan (2009), and Rahman and Devanbu (2013), who investigated relationships between software evolution measures and defect prediction. Nevertheless, the fictional dataset does not establish which feature groups would provide the greatest predictive contribution. A genuine experiment should conduct feature ablation and evaluate structural and historical information separately before interpreting their relative importance.

6.2 Machine Learning and Automated Defect Risk Classification

Machine learning classifiers provide alternative mechanisms for representing relationships between software metrics and historical defect outcomes. Logistic Regression estimates class probabilities through a specified statistical function, while Random Forest aggregates predictions from multiple decision trees. Gradient Boosting develops an ensemble through sequential optimization. These methods differ in their representation of nonlinear relationships, sensitivity to feature interactions, and computational requirements. Their suitability must therefore be evaluated using representative source code records and consistent assessment procedures.

The hypothetical comparison presents one possible scenario in which Gradient Boosting produces favorable precision, recall, and F1-score values. However, these values cannot demonstrate empirical superiority because the models were not trained or tested. A completed investigation would need to examine performance across independent software repositories and report uncertainty around comparative differences. It should also determine whether apparent improvements reflect meaningful source code relationships or project-specific patterns. Predictive accuracy should not be interpreted as proof that the model understands the causes of software defects.

6.3 Defect Prediction and Software Quality Assurance Integration

Automated defect prediction may support software quality assurance by helping developers identify files that warrant additional testing or code review. A risk score can contribute to prioritization when inspection resources are limited, particularly in large repositories containing numerous components.

However, a predicted defect-prone file is not equivalent to a confirmed software error. The model identifies an association with historical defect patterns, whereas actual defect confirmation requires appropriate testing, inspection, or other technical evidence.

The practical usefulness of defect prediction also depends on the cost of investigating false-positive results and the consequences of missing genuine defects. A model that flags many files may achieve high recall while creating additional inspection effort. Conversely, conservative predictions may reduce unnecessary reviews but overlook important problems. The proposed framework therefore emphasizes developer-centered evaluation and workload-sensitive thresholds. Future empirical research should examine whether predictive prioritization improves defect discovery, inspection efficiency, and software maintenance outcomes compared with existing quality assurance practices.

7. Challenges

7.1 Defect Label Quality and Source Code Measurement Consistency

Software defect labels are frequently constructed by connecting issue reports, version-control histories, and corrective source code changes. These records may contain incomplete information, inconsistent issue classifications, or incorrect links between defects and files. A change recorded as a bug fix may instead represent a feature modification or refactoring activity. Consequently, automatically generated labels can include false associations and missing defect information. Such errors may affect both classifier training and the validity of reported evaluation results.

Source code measurement introduces additional difficulties because software metrics depend on parsing rules, programming language characteristics, and tool configuration. Different static analysis tools may generate different warning categories or calculate complexity measures using different conventions. Code generation and formatting practices can also affect selected metrics. The proposed framework therefore emphasizes consistent extraction procedures and documented feature definitions. A genuine investigation should verify defect labels and evaluate measurement reliability before interpreting associations between source code characteristics and defect outcomes.

7.2 Project Heterogeneity and Model Generalization

Software repositories differ in architectural structure, development methodology, programming conventions, dependency management, and maintenance history. These differences can influence the distributions of source code metrics and the relationships between predictive features and defects. A classifier trained on one collection of projects may therefore perform differently when applied to an unfamiliar repository. Consequently, project-independent testing provides a stronger assessment of transferability than evaluation restricted to closely related source files from the same project.

Model generalization may also be affected by differences in project size, programming language version, static analysis rules, and defect reporting practices. Even within the same programming language, two development teams may use substantially different coding patterns. These differences can reduce the usefulness of previously learned predictive relationships. The proposed methodology therefore emphasizes repository-level validation and transparent reporting of project characteristics. Future research should examine whether model adaptation or additional project-specific training information improves performance without introducing evaluation bias.

7.3 Class Imbalance, Information Leakage, and Model Interpretability

Defect prediction commonly involves unequal numbers of source files with and without recorded defects. Such imbalance can influence classification thresholds and produce misleading aggregate accuracy. A model may classify most non-defective files correctly while failing to identify an important proportion of defect-linked components. Consequently, precision, recall, and F1-score provide important complementary evaluation information. Resampling and class weighting may be considered during training, but these procedures require careful validation and should not alter the independent test distribution.

Information leakage presents a further challenge when features contain information recorded after the prediction snapshot or when closely related code appears across training and testing datasets. Such leakage can produce unrealistically favorable performance estimates. Model interpretability is also important because developers need understandable reasons for prioritizing particular files. However, feature importance scores describe predictive associations and should not be interpreted as causal explanations. The proposed framework emphasizes timestamp controls, project-independent evaluation, transparent feature definitions, and appropriate interpretation of model outputs.

8. Conclusion

Machine learning and source code analysis provide complementary methodological approaches for identifying software components associated with elevated defect risk. The present paper developed a hypothetical comparative framework involving rule-based static analysis, Logistic Regression, Random Forest, and Gradient Boosting. Its principal focus was the relationship between source code metrics, historical development characteristics, defect labels, and predictive classification performance. The proposed methodology demonstrates how automated analysis could support the prioritization of software quality assurance activities through structured feature extraction and independent evaluation.

The illustrative tables demonstrate possible procedures for reporting defect distributions and comparing machine learning classification outcomes. However, all numerical values were manually constructed and do not represent actual software repositories or completed computational experiments. The hypothetical comparison assigns the highest F1-score to Gradient Boosting, but this ranking cannot establish empirical superiority. Meaningful conclusions would require authentic source code datasets, verified defect histories, implemented analysis procedures, independent testing, and suitable uncertainty assessment.

Overall, the study identifies defect-label reliability, source code measurement consistency, project heterogeneity, class imbalance, information leakage, and model interpretability as important considerations in automated software defect prediction. Future empirical investigations should evaluate these factors across representative software repositories and development environments. Machine learning may support quality assurance when predictive information is integrated with code review and software testing. Nevertheless, reliable software defect detection must remain grounded in validated technical evidence rather than predictive scores alone.

References

1. Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL), Article 40, 1–29. <https://doi.org/10.1145/3290353>
2. Arisholm, E., Briand, L. C., & Johannessen, E. B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1), 2–17. <https://doi.org/10.1016/j.jss.2009.06.018>
3. Bird, C., Nagappan, N., Devanbu, P., Gall, H., & Murphy, B. (2011). Don't touch my code! Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (pp. 4–14). ACM. <https://doi.org/10.1145/2025113.2025119>
4. Catal, C. (2011). Software fault prediction: A literature review and current trends. *Expert Systems with Applications*, 38(4), 4626–4636. <https://doi.org/10.1016/j.eswa.2010.10.024>
5. Chen, Z., Cao, Y., Wang, H., Chen, T., & Zhang, Y. (2022). Deep learning-based software defect prediction via semantic key features of source code—Systematic survey. *Mathematics*, 10(17), Article 3120. <https://doi.org/10.3390/math10173120>
6. Dehaghani, S. M. H., & Hajrahimi, N. (2013). Which factors affect software projects maintenance cost more? *Acta Informatica Medica*, 21(1), 63–66.
7. Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6), 1276–1304. <https://doi.org/10.1109/TSE.2011.103>
8. Hejazi Dehaghani, S. M., & Hajrahimi, N. (2013). Which factors affect software projects maintenance cost more? *Acta Informatica Medica*, 21(1), 63–66.
9. Hoang, T., Khanh Dam, H., Kamei, Y., Lo, D., & Ubayashi, N. (2019). DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories* (pp. 34–45). IEEE. <https://doi.org/10.1109/MSR.2019.00017>
10. Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6), 757–773. <https://doi.org/10.1109/TSE.2012.70>
11. Kim, S., Zimmermann, T., Whitehead, E. J., Jr., & Zeller, A. (2007). Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering* (pp. 489–498). IEEE. <https://doi.org/10.1109/ICSE.2007.66>
12. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*. Internet Society. <https://doi.org/10.14722/ndss.2018.23158>
13. Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27, 504–518. <https://doi.org/10.1016/j.asoc.2014.11.023>
14. Nam, J., Pan, S. J., & Kim, S. (2013). Transfer defect learning. In *Proceedings of the 2013 IEEE/ACM 35th International Conference on Software Engineering* (pp. 382–391). IEEE. <https://doi.org/10.1109/ICSE.2013.6606584>

15. Ray, B., Hellendoorn, V., Godhane, A., Tu, Z., Bacchelli, A., & Devanbu, P. (2016). Deep learning vs. traditional learning approaches for software defect prediction. In Proceedings of the 2016 IEEE/ACM 12th Working Conference on Mining Software Repositories (pp. 313–323). ACM. <https://doi.org/10.1145/2901739.2901744>
16. Shepperd, M., Bowes, D., & Hall, T. (2014). Researcher bias: The use of machine learning in software defect prediction. IEEE Transactions on Software Engineering, 40(6), 603–616. <https://doi.org/10.1109/TSE.2014.2322358>
17. Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2016). An empirical comparison of model validation techniques for defect prediction models. IEEE Transactions on Software Engineering, 43(1), 1–18. <https://doi.org/10.1109/TSE.2016.2584050>
18. Van Dinter, R., Catal, C., Giray, G., & Tekinerdogan, B. (2023). Just-in-time defect prediction for mobile applications: Using shallow or deep learning? Software Quality Journal, 31, 1281–1302. <https://doi.org/10.1007/s11219-023-09629-1>
19. Wang, S., Liu, T., & Tan, L. (2016). Automatically learning semantic features for defect prediction. In Proceedings of the 38th International Conference on Software Engineering (pp. 297–308). ACM. <https://doi.org/10.1145/2884781.2884804>
20. Zhang, F., Zheng, Q., Zou, Y., & Hassan, A. E. (2016). Cross-project defect prediction using a connectivity-based unsupervised classifier. In Proceedings of the 38th International Conference on Software Engineering (pp. 309–320). ACM. <https://doi.org/10.1145/2884781.2884839>