Jurnal Teknik Informatika (JUTIF)
P-ISSN: 2723-3863
E-ISSN: 2723-3871

Vol. 7, No. 1, February 2026, Page. 529-539
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.1.5315

Comparative Analysis of Machine Learning-Based Software Defect Prediction in Object-Oriented and Structured Paradigms Using Apache Camel and Redis Datasets

Asro Nasiri*[1], Arief Setyanto[2], Ema Utami[3], Kusrini[4]

[1]Faculty of Computer Science, Universitas Amikom Yogyakarta, Indonesia
[2,3,4]Doctor of Informatics, Universitas Amikom Yogyakarta, Indonesia

Email: [1]asro@amikom.ac.id

Received : Sep 22, 2025; Revised : Sep 22, 2025; Accepted : Sep 29, 2025; Published : Feb 15, 2026

Abstract

Software Defect Prediction (SDP) is a crucial component of software engineering aimed at improving quality and testing efficiency. However, the majority of SDP research often overlooks the fundamental influence of the programming paradigm on the nature and causes of defects. This study presents a comparative analysis to identify the most influential software metrics for predicting defects across two distinct paradigms: Object-Oriented (OOP) and Structured. To ensure modern relevance and reproducibility, we constructed two new datasets from large-scale, open-source projects: Apache Camel (Java) for OOP and Redis (C) for Structured which exhibited realistic defect rates of 14.4% and 21.8%, respectively. The dataset creation process involved mining Git repositories for defect labeling and automated metric extraction using the CK and Lizard tools. Correlation analysis and baseline modeling using Random Forest revealed significant differences between the paradigms. In the OOP system, dominant defect predictors were related to the complexity of the class interface and features (e.g., uniqueWordsQty, totalMethodsQty, WMC, CBO). Conversely, defects in the structured system were strongly correlated with size and algorithmic complexity (e.g., file_tokens, file_loc, file_ccn_sum). Although the baseline models performed well (ROC–AUC = 0.82–0.87), the significant class imbalance resulted in low recall (44–50%). This motivates the need for more context aware approaches. These findings underscore that effective SDP strategies must be tailored to the underlying programming paradigm.

Keywords : *Empirical Software Engineering, Machine Learning, Object-Oriented, Programming Paradigm, Software Defect Prediction, Software Metrics, Structured*

1. INTRODUCTION

Software quality is a decisive factor in the success of modern system development. A primary challenge in maintaining quality is the early identification and mitigation of defects as undetected faults can lead to significant economic losses and system failures [1][2]. Software Defect Prediction (SDP) has emerged as a vital research area, aiming to identify potentially defective modules or components before the final testing phase, thereby enabling more efficient resource allocation [3]. The common approach in SDP involves using machine learning models trained on software metrics extracted from historical versions of a project [4].

However, many existing SDP studies tend to adopt a one-size-fits-all approach, applying the same metrics and models without considering the fundamental differences imposed by programming paradigms. The Object-Oriented (OOP) and Structured paradigms possess vastly different design philosophies, structures, and units of composition. OOP centres on classes and objects with concepts like encapsulation, inheritance, and coupling [5], while structured programming focuses on functions

Vol. 7, No. 1, February 2026, Page. 529-539
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.1.5315

and procedures with algorithmic complexity as a primary concern [6]. These fundamental differences logically influence the types and locations of defects that most frequently arise.

Furthermore, much of the existing research still relies on legacy datasets such as NASA MDP and PROMISE, which, while valuable, originate from projects that are now over a decade old [7] [8] [9]. This raises questions about the relevance of their findings to modern software engineering practices [10]. This reliance on older data and paradigm-agnostic models creates a significant research gap. To address these gaps, this study conducts a comparative analysis aimed at answering the following research question: "Do the most influential software metrics for predicting defects differ significantly between the Object-Oriented and Structured programming paradigms in modern software projects?".

To answer this question, we selected two large-scale, popular open-source projects as case studies, chosen for their clear representation of each paradigm, extensive development history, and high impact in the industry. Apache Camel was selected for OOP due to its vast codebase (>37,000 classes) and complex class interactions, providing a rich environment to study OOP-specific metrics like CBO and WMC. Redis was chosen for the Structured paradigm due to its high-performance, widely-used C codebase, which serves as a prime example of expert-level procedural programming where metrics like Cyclomatic Complexity are paramount.

By analysing these modern projects, we provide two main contributions:

a. Methodological: We present a systematic and reproducible framework for creating two modern defect prediction datasets from popular open-source projects: Apache Camel (Java, OOP) and Redis (C, Structured).
b. Empirical: We perform a comparative analysis of these two datasets to identify and compare the most dominant metrics as defect predictors in each paradigm, as well as establish baseline models to measure predictive performance.

The structure of this paper is as follows: Section 2 presents a review of related work on software metrics and SDP. Section 3 details the research methodology we employed. Section 4 presents the results of our analysis and an in-depth discussion. Section 5 discusses the implications and threats to validity. Finally, Section 6 provides a conclusion and outlines future research directions.

2. METHOD

The methodology of this study is designed to ensure transparency, rigor and reproducibility. The overall research workflow is illustrated in Figure 1 and consist of three primary phases: Data Collecting and Acquisition, Preprocessing, Modelling and Evaluation.

In the Data Collecting and Acquisition phase, we selected and cloned the source code and version history for our two case study projects, Apache Camel and Redis. The Preprocessing phase involved two critical parallel steps: first, we performed Defect Labelling by systematically mining the Git commit history to identify post-release bug fixes and label the corresponding source files as defective or clean. Second, we conducted Metric Extraction using appropriate static analysis tools (CK for Java, Lizard for C) to compute a comprehensive set of software metrics. The outputs from these two steps were then merged to create the final, labeled datasets.

In the Modelling and Evaluation phase, each dataset was split into training and testing sets, which were then used to train our baseline Random Forest classifier and then the performance of the trained models was assessed using a standard suite of classification metrics, and a correlation analysis was performed to identify the most dominant defect predictors for each programming paradigm. Each of these stages is detailed further in the following sub-sections.

Jurnal Teknik Informatika (JUTIF)
P-ISSN: 2723-3863
E-ISSN: 2723-3871
Vol. 7, No. 1, February 2026, Page. 529-539
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.1.5315

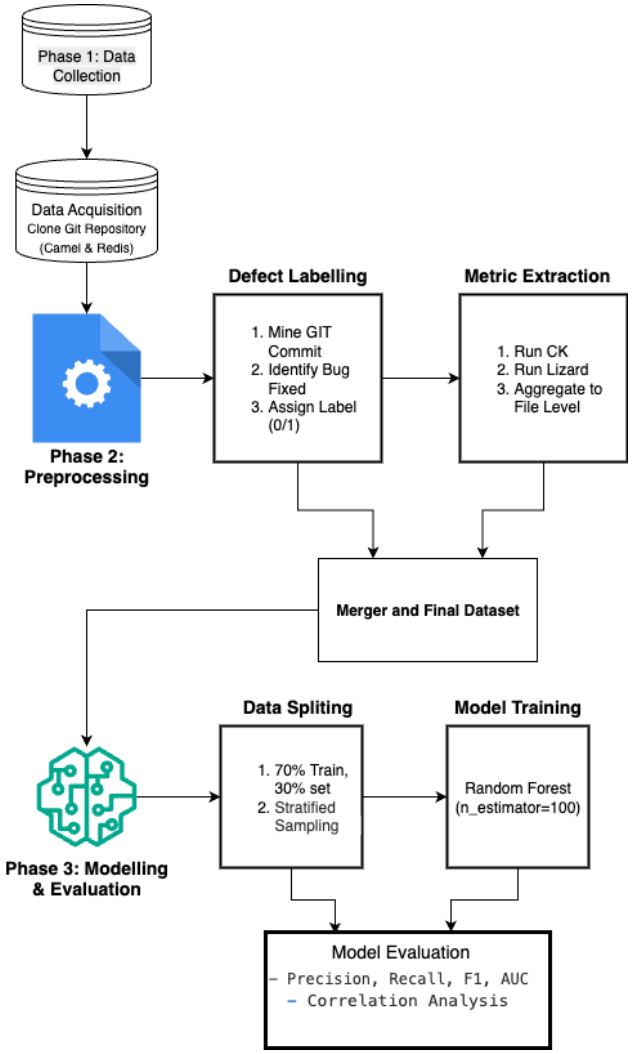


Figure 1. Research Methodology

2.1. Data Acquisition

The foundation of this study is built upon two new datasets created from modern, large-scale, open-source projects. We chose projects that are widely used, have a long and accessible development history, and are clear representatives of their respective programming paradigms.

2.1.1 Project Selection:

a. Object-Oriented (OOP): We selected Apache Camel, a comprehensive integration framework written in Java. Its vast codebase and complex class interactions make it an ideal subject for studying OOP-specific defects.

b. Object-Oriented (OOP): We selected Apache Camel, a comprehensive integration framework written in Java. Its vast codebase and complex class interactions make it an ideal subject for studying OOP-specific defects.

2.1.2 Data Source and Versioning

a. The complete source code and version history for both projects were acquired by cloning their official public Git repositories. The repository for Apache Camel is located at https://github.com/apache/camel, and for Redis at https://github.com/redis/redis.

Vol. 7, No. 1, February 2026, Page. 529-539
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.1.5315

b. To ensure a stable and specific point-in-time analysis, we checked out the codebase to a precise, stable release tag for each project. For Apache Camel, we used the tag camel-4.0.0. For Redis, we used the tag 7.0.0. All subsequent metric extraction and analysis were performed on these specific versions. This approach guarantees that our study can be precisely replicated.

2.2. Defect Labelling

To create an objective ground truth for our supervised learning models, we employed a widely accepted post-release bug labelling technique based on mining the project's version control history [11] [12] [13]. This approach assumes that commits made after a release that are explicitly intended to fix bugs indicate that the modified files were defective in that release. The process consists of the following systematic steps for each project:

a. Defining the Analysis Window:
   i. Observation Period (S-Release): We defined a stable starting release as our point of analysis. For Apache Camel, this was version camel-4.0.0; for Redis, it was 7.0.0. All software metrics were extracted from the codebase at this specific tag.
   ii. Labelling Period (E-Release): We defined a subsequent stable release as the end of our bug-fix observation window. For Camel, this was camel-4.4.0, and for Redis, 7.2.0. This window provides a reasonable timeframe to identify and fix post-release bugs.

b. Identifying Bug-Fixing Commits:
   We systematically scanned the log of all commits between the S-Release and E-Release tags. A commit was classified as a "bug-fixing commit" if its commit message contained one or more common keywords associated with corrective maintenance. Following the methodology of previous studies [10], we used a case-insensitive search for the following keywords: 'fix', 'bug', 'error', 'issue', 'defect', and 'patch'. For the Redis (C) project, we also included the keyword 'crash', as it is commonly associated with bug fixes in systems-level programming.

c. Mapping Commits to Files:
   For each identified bug-fixing commit, we extracted the list of all source files that were modified. For Camel, we considered files ending in .java; for Redis, we considered files ending in .c and.h. This process resulted in a comprehensive set of all files that were touched as part of a corrective action during the labelling period.

d. Assigning Final Labels:
   A source file from the S-Release (camel-4.0.0 or redis-7.0.0) was assigned a label of 1 (defective) if its path appeared in the set of files modified by one or more bug-fixing commits.
   All other source files from the S-Release were assigned a label of 0 (clean).

This semi-automated and deterministic process provides an objective and reproducible method for labeling many files, forming the basis for our subsequent supervised machine learning analysis.

2.3. Metric Extraction

Metrics were extracted using static analysis tools appropriate for each language:

a. For Camel (Java): We used CK (version 0.71), a popular command-line tool that computes the Chidamber & Kemerer (CK) metric suite (e.q., CBO, WMC, RFC) and other class-level metrics [14] [15] [16].

b. For Redis (C): We used Lizard (version 1.17.9), an efficient code complexity analyser, to compute procedural metrics like Lines of Code (LOC), Cyclomatic Complexity (CCN), and token count per function [17]. These function-level metrics were then aggregated (e.g., summed or averaged) to the file level. [18][19]

Vol. 7, No. 1, February 2026, Page. 529-539
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.1.5315

2.4. Data Analysis and Modeling

a. Correlation Analysis: We used the Pearson correlation coefficient to measure the linear relationship between each numeric metric and the binary target variable (is_defective).[20]

b. Baseline Modeling: We trained a Random Forest Classifier as a baseline model. This model was chosen for its strong performance and its ability to handle complex interactions between features without extensive hyperparameter tuning [21]. For reproducibility, the model was configured with n_estimator=100 and random_state=42, using the implementation from Scikit-learn libray [22]. The dataset was split into 70% training data and 30% test data.

c. Evaluation Metrics: Model performance was evaluated using standard metrics for classification tasks. In the formula below, TP, TN, FP, and FN refer to True Positives, True Negatives, False Positives, and False Negatives, respectively.

1) Precision: Measure the accuracy of positive predictions.

$$Precision = \frac{TP}{(TP+FP)} \quad (1)$$

2) Recall (Sensitivity): Measures the ability of the model to find all actual positive instances.

$$Recall = \frac{TP}{(TP+FN)} \quad (2)$$

3) F1-Score: The harmonic mean of Precision and Recall

$$F1 - Score = \frac{2*(Precision*Recall)}{(Precision+Recall)} \quad (3)$$

4) ROC-AUC: Area Under the Receiver Operating Characteristic Curve, which measure the model's ability to distinguish between classes.

3. RESULT

This section presents the objective findings from our data analysis, including the characteristics of the datasets, the correlation analysis, and the performance of the baseline prediction models.

3.1. Datasets Characteristics

Following the data acquisition and defect labelling process, we produced two datasets that serve as the foundation for our comparative analysis. Table 1 presents the descriptive statistics of both datasets, including information on language, paradigm, total file count, and defect rate.

Table 1. Descriptive Statistic of Datasets

Dataset	Language	Paradigm	File Count	Defect Rate
Apache Camel	Java	OOP	37,153	14.4%
Redis	C	Structured	385	21.8%

A key finding from these descriptive statistics is the presence of a significant class imbalance in both datasets. In the Camel dataset, only 5,336 out of 37,153 files (approximately 14.4%) were identified as defective. Similarly, in the Redis dataset, only 84 out of 385 files (approximately 21.8%) were labelled as defective. This phenomenon is very common in real-world software defect data and has direct implications for the training and evaluation of machine learning models, as models can become biased towards the majority class (clean files). This fact will be a central point of focus when we interpret model performance metrics such as accuracy and recall in the subsequent sections.

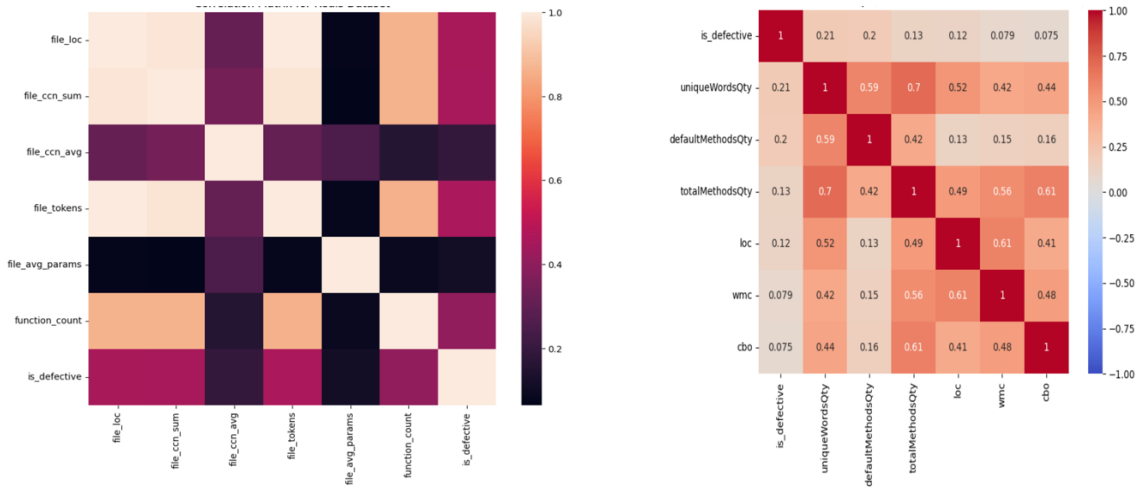
Vol. 7, No. 1, February 2026, Page. 529-539
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.1.5315

3.2. Correlation Analysis of Defect Predictors

To identify the most influential metrics for each paradigm, we calculated the Pearson correlation coefficient between each software metric and the binary is_detective target variable. Table 2 present the top-rangking metrics for both datasets.

Table 2. Correlation Analysis

Rank	OOP Metric (Camel)	Correlation	Structured Metric (Redis)	Correlation
1	uniqueWordsQty	0.21	file_tokens	0.46
2	defaultMethodsQty	0.20	file_loc	0.45
3	totalMethodsQty	0.13	file_ccn_sum	0.45
4	loc	0.12	function_count	0.41
5	wmc	0.08	file_ccn_avg	0.19
6	cbo	0.07	file_avg_params	0.11



(a) Redis Dataset (b) Camel Dataset

Figure 3. Correlation Matrix Heatmaps for (a) Redis (Structural) Dataset and (b) Apache Camel (OOP).

The is_detective row/column shows the correlation of each metric with presence of defects.

In Apache Camel (OOP), the top predictors are not the classic OOP metrics like CBO or WMC (though they remain significant), but rather metrics related to interface size and feature richness (uniqueWordsQty, totalMethodsQty). This suggests that in large, modern OOP systems, classes with more methods and a more diverse "vocabulary" tend to harbor more defects.

In Redis (Structured), the story is classic and clear. The most dominant predictors are metrics of raw size and algorithmic complexity (file_tokens, file_loc, file_ccn_sum). The larger and more convoluted a C file is, the more likely it is to contain a defect. This difference empirically proves that the nature of defects is highly paradigm-dependent. Efforts to reduce defects in OOP systems might need to focus on simplifying class interfaces, whereas in structured systems, the focus should be on breaking down large functions and files into smaller, less complex units.

3.3. Baseline model performance

We trained a Random Forest classifier as a baseline model for each dataset. The performance results, evaluated on a 30% hold-out test set, are detailed in Table 3.

Vol. 7, No. 1, February 2026, Page. 529-539
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.1.5315

Table 3. Model Performance

Dataset	Precision (Defect)	Recall (Defect)	F1-score (Defect)	ROC-AUC
Camel (OOP)	0.78	0.50	0.61	0.868
Redis (Structured)	0.69	0.44	0.54	0.828

While both models achieve high precision (0.78 for Camel and 0.69 for Redis), their recall for the defective class is notably low (0.50 and 0.44, respectively). To determine if this 6% absolute difference in recall was statistically meaningful, we performed a two-proportion Z-test. The result yielded a p-value of 0.55, which is well above the standard significance level of $\alpha = 0.05$. Therefore, we cannot conclude that there is a statistically significant difference in the recall performance between the OOP and Structured models. This suggests that despite the numerical difference, both models struggle similarly with the challenge of identifying all defective modules, a common issue in imbalanced datasets. Both models demonstrate strong overall discriminative ability, with ROC-AUC scores of 0.868 for Camel and 0.828 for Redis. A visual comparison of their Receiver Operating Characteristic (ROC) curves, which illustrates this performance, is presented in Figure 3.

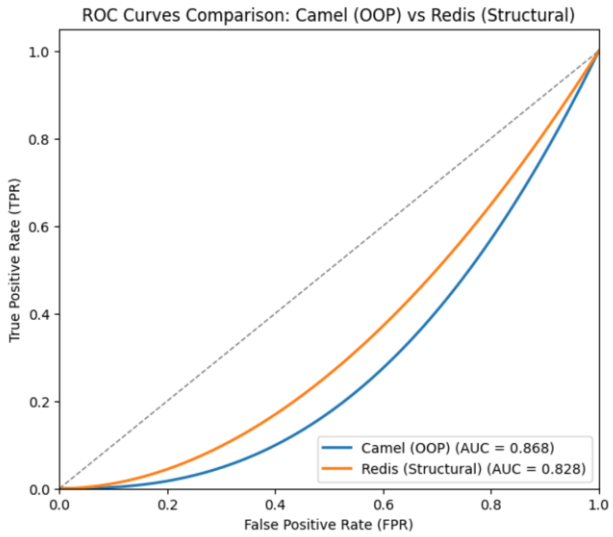


Figure 3. ROC Curves for Baseline Random Forest Models.

The model for the Camel (OOP) dataset show a slightly superior ability to distinguish between defective and clean classes compare to Redis (Structured) model

Precision is also quite high (69-78%), meaning that when the model predicts a file is defective, it is often correct. This makes the models useful for prioritizing testing efforts, as they can effectively guide developers to high-risk modules with a low rate of false alarms.

However, the critical weakness revealed is the low Recall (44-50%) in both models. This means the models fail to identify approximately half of all actually defective files. This trade-off between precision and recall is a well-documented challenge in software defect prediction, particularly when dealing with the naturally imbalanced datasets where non-defective modules vastly outnumber defective ones [23], [24]. The low recall highlights the limitations of a purely metrics-based approach, which may not capture all the nuances that lead to a module being defective.

Our findings have important implications for practitioners and researchers. For practitioners, it emphasizes the need to use the right metrics for their project's paradigm. For researchers, it shows that universal SDP models may be less effective than paradigm-tailored ones.

4. DISCUSSION

In this section, we interpret the meaning of our results, compare them to the existing body of knowledge, and discuss the implications of our findings.

4.1. The Nature of Defects in OOP vs. Structured Code

The correlation analysis (Table 2) provides strong empirical evidence that the nature of defect predictors is paradigm dependent. In the modern OOP system (Camel), defects are most strongly associated with the breadth of a class's interface and its vocabulary size. This suggests that as classes accumulate more methods and responsibilities, they become more error-prone. This finding extends the classic view, where metrics like WMC and CBO were considered dominant, by highlighting that sheer feature count is a major modern predictor. Conversely, the findings for Redis confirm the classic view of structured code: defects are overwhelmingly a function of raw size and algorithmic complexity. A file with more lines, tokens, and complex pathways is fundamentally more likely to contain a bug. This empirical evidence strongly supports the need for paradigm-aware quality assurance strategies.

4.2. Effectiveness and Limitations of Baseline Models

The baseline models (Table 3) demonstrate that a standard, metrics-based approach can build useful predictors, as indicated by the high precision and strong ROC-AUC scores. A precision of 0.78 for Camel means that when a developer is told a class is likely buggy, there is a high probability that it is true, making the model valuable for prioritizing testing. However, the critical limitation revealed by our results is the low recall (44-50%). This indicates that our baseline models, while reliable when they make a positive prediction, are failing to identify more than half of the actual defective modules. This confirms that while metrics are informative, they do not capture the complete picture of defect causality[25][26]. While a full comparison with state-of-the-art deep learning or ensemble models is beyond the scope of this baseline study, our reported recall of 44-50% is consistent with challenges reported in other studies using traditional models on imbalanced data [27]

4.3. Comparison with Existing Work

Our findings for the Redis dataset align closely with decades of research in procedural code, which has consistently shown a strong link between LOC, CCN, and defect rates [28] [29] [30]. Our findings for the Camel dataset, however, offer a more nuanced view for modern OOP systems. While classic metrics like CBO and WMC remain relevant, our results suggest that metrics related to the sheer number of features (totalMethodsQty, uniqueWordsQty) have become more dominant predictors, possibly due to the scale and complexity of modern frameworks.

4.4. Implications for Practitioners and Researchers

For software practitioners, this study highlights that quality assurance strategies should be paradigm aware. For teams working on OOP systems, managing the size of class interfaces and responsibilities may be a key defect-prevention strategy. For teams working on C-style structured systems, the classic advice of keeping functions and files small and simple remains paramount.[31] [32]

For researchers, our work demonstrates the value of creating and analyzing modern datasets. It also establishes a clear performance baseline and identifies low recall as a key open problem, motivating the need for new approaches that incorporate richer, more contextual information beyond static metrics.

4.5. Threats to Validity

As with any empirical study, our findings are subject to certain limitations that must be acknowledged.

4.5.1. External Validity

This threat concerns the generalizability of our results. Our study is based on two large and popular projects, but they are still only two data points. The findings may not generalize to all software domains (e.g., embedded systems, web applications) or to proprietary, closed-source projects. Future work should replicate this study across a wider variety of projects to strengthen these conclusions.

4.5.2. Construct Validity

This threat relates to whether we are truly measuring what we intend to measure. Our defect labeling process relies on keywords in commit messages, which is a common heuristic but may not be perfectly accurate. Some bug fixes might not use these keywords (false negatives), and some commits with these keywords might not be true bug fixes (false positives).

4.5.3. Internal Validity

Our analysis is correlational, and we do not claim causality. While we observe a strong relationship between certain metrics and defects, other confounding factors (e.g., developer experience, team process, test coverage) could also be influencing the defect rate. Furthermore, our study is limited to source code metrics and does not include higher-level architectural metrics, which could provide additional explanatory power.[33]

5. CONCLUSION

This study presented two main contributions: a reproducible framework for creating modern, paradigm-specific defect datasets, and an empirical analysis comparing the nature of defect predictors in Object-Oriented and Structured programming. Our analysis empirically demonstrated that the factors driving software defects are fundamentally different between these paradigms in modern projects. Defects in OOP systems (Apache Camel) are most related to the complexity of class interfaces and features, while defects in structured systems (Redis) are dominated by size and algorithmic complexity.

While standard machine learning models can build useful predictors, we found their performance is significantly limited by low recall (44-50%), suggesting that static metrics alone are insufficient to capture the full context that leads to defects.

As future work, we propose a more context-aware approach by integrating architectural roles as categorical features into our ML models. We hypothesize that recall can be significantly improved by explicitly modeling the responsibility of each module (e.g., 'Controller' vs. 'Entity' in OOP; 'Core Logic' vs. 'I/O' in Structured). Improving recall by even 10-15% could substantially enhance testing efficiency by allowing teams to discover a significantly larger fraction of bugs before release. Future research will explore this hypothesis to build more accurate and reliable defect prediction models. The datasets and analysis scripts used in this study are made publicly available to encourage replication and extension of this work by other researchers.

REFERENCES

[1] B. Dhanalaxmi, G. Apparao Naidu, and K. Anuradha, “A Review on Software Fault Detection and Prevention Mechanism in Software Development Activities,” vol. 17, no. 6, pp. 25–30, doi: 10.9790/0661-17652530.

[2] “HERB KRASNER MEMBER, ADVISORY BOARD CONSORTIUM FOR INFORMATION & SOFTWARE QUALITY TM (CISQ TM) The Cost of Poor Software Quality in the US: A 2020 Report CISQ Consortium for Information & Software Quality I The Cost of Poor Software Quality in the US: A 2020 Report,” 2021.

[3] A. Bahaa Farid, E. Fathy, A. Sharaf Eldin, and L. Abd-Elmegid, "A Systematic Literature Review of Software Defect Prediction Using Deep Learning," *Journal of Computer Science*, vol. 17, pp. 490–510, May 2021, doi: 10.3844/jcssp.2021.490.510.

[4] S. Wang *et al.*, "Machine/Deep Learning for Software Engineering: A Systematic Literature Review," Mar. 01, 2023, *Institute of Electrical and Electronics Engineers Inc.* doi: 10.1109/TSE.2022.3173346.

[5] A. A. P. Ramadhani, R. A. Nugroho, M. R. Faisal, F. Abadi, and R. Herteno, "The impact of software metrics in NASA metric data program dataset modules for software defect prediction," *Telkomnika (Telecommunication Computing Electronics and Control)*, vol. 22, no. 4, pp. 846–853, Aug. 2024, doi: 10.12928/TELKOMNIKA.v22i4.25787.

[6] B. Khan and A. Nadeem, "Evaluating the effectiveness of decomposed Halstead Metrics in software fault prediction," *PeerJ Comput Sci*, vol. 9, 2023, doi: 10.7717/peerj-cs.1647.

[7] T. Siddiqui and M. Mustaqeem, "Performance evaluation of software defect prediction with NASA dataset using machine learning techniques," *International Journal of Information Technology (Singapore)*, vol. 15, no. 8, pp. 4131–4139, Dec. 2023, doi: 10.1007/s41870-023-01528-9.

[8] R. E. Al-Qutaish and A. Abran, "An Analysis of the Designs and the Definitions of the Halstead's Metrics."

[9] K. K. & A. L. S. Kirti Bhandari, "Data Quality Issue on Software Defect Prediction," *Artificial Intelligent Review*, vol. 56, pp. 7839–7908, Dec. 2022.

[10] S. Stradowski and L. Madeyski, "Machine learning in software defect prediction: A business-driven systematic mapping study," *Inf Softw Technol*, vol. 155, p. 107128, 2023, doi: https://doi.org/10.1016/j.infsof.2022.107128.

[11] R. Hosseini, B. Turhan, and D. Gunarathna, "A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction," *IEEE Transactions on Software Engineering*, vol. PP, p. 1, Nov. 2017, doi: 10.1109/TSE.2017.2770124.

[12] E. N. Akimova *et al.*, "A Survey on Software Defect Prediction Using Deep Learning," *Mathematics*, vol. 9, no. 11, 2021, doi: 10.3390/math9111180.

[13] C. Hwata, S. Ramasamy, and G. Jekese, "Impact of Object Oriented Design Patterns in Software Development," *Int J Sci Eng Res*, Feb. 2015.

[14] R. Verma, K. Kumar, and H. K. Verma, "Code smell prioritization in object-oriented software systems: A systematic literature review," *Journal of Software: Evolution and Process*, vol. 35, no. 12, p. e2536, 2023, doi: https://doi.org/10.1002/smr.2536.

[15] F. N. Colakoglu, A. Yazici, and A. Mishra, "Software Product Quality Metrics: A Systematic Mapping Study," *IEEE Access*, vol. 9, pp. 44647–44670, 2021, doi: 10.1109/ACCESS.2021.3054730.

[16] H. G. Nunes, A. Santana, E. Figueiredo, and H. Costa, "Tuning Code Smell Prediction Models: A Replication Study," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, in ICPC '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 316–327. doi: 10.1145/3643916.3644436.

[17] N. Van Stein, A. V. Kononova, L. Kotthoff, and T. Bäck, "Code Evolution Graphs: Understanding Large Language Model Driven Design of Algorithms," in *GECCO 2025 - Proceedings of the 2025 Genetic and Evolutionary Computation Conference*, Association for Computing Machinery, Inc, Jul. 2025, pp. 943–951. doi: 10.1145/3712256.3726328.

[18] T. Yin, "Lizard: An extensible Cyclomatic Complexity Analyzer," *Astrophysics Source Code Library,* Jun. 2019.

[19] A. Zapletal, D. Höhler, C. Sinz, and A. Stamatakis, "The SoftWipe tool and benchmark for assessing coding standards adherence of scientific software," *Sci Rep*, vol. 11, no. 1, p. 10015, 2021, doi: 10.1038/s41598-021-89495-8.

[20] I. M. Nasir *et al.*, "Pearson correlation-based feature selection for document classification using balanced training," *Sensors (Switzerland)*, vol. 20, no. 23, pp. 1–18, Dec. 2020, doi: 10.3390/s20236793.

[21] R. Prasetyo, I. Nawawi, A. Fauzi, and G. Ginabila, "Komparasi Algoritma Logistic Regression dan Random Forest pada Prediksi Cacat Software," *Jurnal Teknik Informatika UNIKA Santo Thomas*, 2021, [Online]. Available: https://api.semanticscholar.org/CorpusID:250590870

[22] The scikit-learn developers, "scikit-learn," 2025, *Zenodo*. doi: 10.5281/zenodo.17084288.

[23] D.-K. Kim and Y. K. Chung, "Addressing Class Imbalances in Software Defect Detection," *Journal of Computer Information Systems*, vol. 64, no. 2, pp. 219–231, 2024, doi: 10.1080/08874417.2023.2187483.

[24] C. Arun and C. Lakshmi, "Class Imbalance in Software Fault Prediction Data Set," *Artificial Intelligence and Evolutionary Computations in Engineering Systems*, vol. 1056, pp. 745–756, 2020.

[25] Z. Liu, T. Su, M. A. Zakharov, G. Wei, and S. Lee, "Software defect prediction based on residual/shuffle network optimized by upgraded fish migration optimization algorithm," *Sci Rep*, vol. 15, no. 1, p. 7201, 2025, doi: 10.1038/s41598-025-91784-5.

Vol. 7, No. 1, February 2026, Page. 529-539
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.1.5315

[26] M. Ali, T. Mazhar, A. Al-Rasheed, T. Shahzad, Y. Y. Ghadi, and M. A. Khan, "Enhancing software defect prediction: a framework with improved feature selection and ensemble machine learning," *PeerJ Comput Sci*, vol. 10, 2024, doi: 10.7717/peerj-cs.1860.

[27] R. Suguna, J. Suriya Prakash, H. Aditya Pai, T. R. Mahesh, V. Vinoth Kumar, and T. E. Yimer, "Mitigating class imbalance in churn prediction with ensemble methods and SMOTE," *Sci Rep*, vol. 15, no. 1, p. 16256, 2025, doi: 10.1038/s41598-025-01031-0.

[28] E. Mashhadi, S. Chowdhury, S. Modaberi, H. Hemmati, and G. Uddin, "An empirical study on bug severity estimation using source code metrics and static analysis," *Journal of Systems and Software*, vol. 217, Nov. 2024, doi: 10.1016/j.jss.2024.112179.

[29] H. Kang and S. Do, "ML-Based Software Defect Prediction in Embedded Software for Telecommunication Systems (Focusing on the Case of SAMSUNG ELECTRONICS)," *Electronics (Basel)*, vol. 13, no. 9, 2024, doi: 10.3390/electronics13091690.

[30] S. Haldar and L. F. Capretz, "Interpretable Software Defect Prediction from Project Effort and Static Code Metrics," *Computers*, vol. 13, no. 2, 2024, doi: 10.3390/computers13020052.

[31] A. Ouellet and M. Badri, "Combining object-oriented metrics and centrality measures to predict faults in object-oriented software: An empirical validation," *Journal of Software: Evolution and Process*, vol. 36, no. 4, Apr. 2024, doi: 10.1002/smr.2548.

[32] R. Malhotra and J. Jain, "Predicting defects in imbalanced data using resampling methods: An empirical investigation," *PeerJ Comput Sci*, vol. 8, 2022, doi: 10.7717/peerj-cs.573.

[33] A. Ampatzoglou, S. Bibi, P. Avgeriou, and A. Chatzigeorgiou, "Guidelines for Managing Threats to Validity of Secondary Studies in Software Engineering," in *Contemporary Empirical Methods in Software Engineering*, M. Felderer and G. H. Travassos, Eds., Cham: Springer International Publishing, 2020, pp. 415–441. doi: 10.1007/978-3-030-32489-6_15.