Review

AI-Driven Software Engineering: A Systematic Review of Machine Learning's Impact and Future Directions

Fred Nyaga *

Posted Date: 2 April 2025

doi: 10.20944/preprints202504.0174.v1

Keywords: machine learning; software engineering,; code generation; error detection; deep learning; model interpretabilit

 doi:10.20944/preprints202504.0174.v1

Review

AI-Driven Software Engineering: A Systematic Review of Machine Learning's Impact and Future Directions

Nyaga Fred. D

Department of Automated Control Systems, National University of Science and Technology "MISiS" (Moscow); m156796@edu.misis.ru

Abstract: This study looks into the expanding importance of machine learning (ML) in software engineering, to provide a thorough evaluation of its applications, categorise existing approaches, and propose prospective areas of future research. As machine learning (ML) continues transforming software development processes, understanding its potential and limitations is critical for success. A systematic literature review was carried out utilising major scientific databases, including IEEE Xplore, ACM Digital Library, Scopus, Arxiv, and Web of Science. After applying strict criteria for inclusion and exclusion to an initial pool of 105 publications, 57 were chosen for further review. The study synthesised concepts from the reviewed literature using both quantitative and qualitative approaches, including thematic coding and statistical analysis of publishing trends. The findings highlight major applications of machine learning in software engineering, including code generation, error detection, and program maintenance. Furthermore, the paper identifies a growing trend in the usage of graph neural networks (GNNs) to analyse code architectures, which are validated by experimental evidence. These achievements demonstrate ML's transformational potential for optimising the software development life cycle. While machine learning offers significant opportunities for automation and optimisation in software engineering, challenges such as low model interpretability, high computation costs, and limited integration into existing workflows remain. Addressing these challenges is important to fully realise ML's potential. Integrating machine learning into traditional programming methodologies, utilising federated learning for privacy-preserving collaboration, and developing interpretable ML models targeted to software engineering roles are all intriguing research avenues.

Keywords: machine learning; software engineering; code generation; error detection; deep learning; model interpretability; systematic review

1. Introduction

Software engineering is a rapidly expanding field that encompasses designing, developing, maintaining, implementing, and evolving software systems in a systematic and controlled manner. Machine learning, a type of artificial intelligence, has gained popularity in recent years due to its ability to analyse massive volumes of data. This has enabled it to be used in software engineering for a variety of purposes, including defect identification, code quality evaluation, requirements analysis, and software project management. In today's SE world, the desire for high-quality, maintainable source code is critical. Software applications are becoming more complicated, making it difficult for developers to write efficient and error-free code.

Researchers like M. Pradel et al. have demonstrated how incorporating machine learning into the software development process may boost productivity and reduce errors. Advances in natural language processing have enabled the application of machine learning in programming. According to D. Wiesel et al., deep learning, a subset of machine learning, is causing a revolution in a variety of

 doi:10.20944/preprints202504.0174.v1

fields such as language processing, image recognition, automatic translation, and text synthesis. Natural language processing techniques can be used in software engineering to perform various text-based tasks such as documentation, error detection, code completion, and code translation, significantly increasing developer productivity and code quality, as demonstrated by Pautsi and colleagues' research. According to Cheng J. S. [9], image processing techniques can be utilised for visual testing, pattern identification, developing interfaces, and enhancing user experience in software systems.

Incorporating these technologies improves software development procedures while likewise enhancing application functionality and usability. Traditional code analysis methods, which focus on precision and logical thinking, have long been used by programmers to understand how programs interact. However, when systems become more complex, these methods become time-consuming, costly, and difficult to implement. Lerman adopted the phrase "software evolution" in the 1970s to characterise the constant modifications and advances in software code. This approach has garnered significant acceptance within the scientific community, owing to the increasing size of software systems. In light of these issues, researchers are looking into developing technologies like machine learning to help manage growth and improve software engineering techniques.

Machine learning-based approaches have shown great promise in terms of optimising software development processes. These approaches excel at solving the complicated challenges of dynamic software development, such as managing system complexities and providing high-quality results. By combining data from many sources, such as repository information, historical change logs, documentation, and issue reports, ML models can offer robust insights and solutions for software engineering problems. Furthermore, analysing existing repositories provides critical insights that could significantly enhance software development, administration, and evolution. The increasing application of machine learning in software engineering provides novel approaches to addressing complicated challenges such as software bugs and security vulnerabilities [1].

The incorporation of machine learning into software engineering provides novel approaches for addressing complex technical challenges, such as software flaws and security vulnerabilities. This systematic review investigates the connection between machine learning (ML) and software engineering (SE) through three major research objectives:

1. Gap Identification. Determine problems and research opportunities in ML applications within SE.
2. Applicability Assessment. Determine the usefulness of ML approaches in enhancing SE processes.
3. Innovation Facilitation. Propose research directions to promote cross-disciplinary collaboration and the incorporation of machine learning into SE processes.

2. Related Works

There is an increasing body of research on the application of machine learning (ML) in software engineering, motivated by its potential to improve the efficiency and effectiveness of software development processes. Traditional techniques, such as rule-based static analysis and human code inspections, usually fail to keep up with the complex nature of current software systems and may miss subtle, context-dependent problems. This section analyzes the existing literature to address our four research questions.

2.1. RQ1: What Are the Main Applications of Machine Learning in Software Engineering?

The literature reveals several significant applications of machine learning in software engineering:

 doi:10.20944/preprints202504.0174.v1

Code Analysis and Bug Detection

Michael Pradel et al. (2021) conducted pioneering studies on software analysis using neural networks and later developed DeepBugs, a machine learning-powered tool [1,2]. This approach achieved 71% accuracy in detecting bugs across JavaScript and Python projects, demonstrating ML's capacity to resolve name-based inconsistencies pervasive in real-world code [1]. Their research showed that neural networks could successfully detect semantic errors such as swapped function arguments (e.g., passing velocity and acceleration in the wrong order) or misused variables (e.g., multiplying price by tax instead of quantity), outperforming previous methods based on syntactic or heuristic rules [1,2,7].

Tian et al. (2015) presented an approach for detecting flaws in real time during code reviews by analyzing changes when developers submit code (e.g., in pull requests) and flagging issues such as security vulnerabilities or logical mistakes [55]. This instant feedback allows teams to address problems early on, reducing the cost and labor required for later improvements.

Software Maintenance

Maintaining the relevance and efficiency of software systems has become a primary priority in the face of rapid technological progress and increasing demand. Hall et al. (2011) found that machine learning algorithms such as decision trees, neural networks, and support vector machines are helpful in predicting software difficulties. These models allow for the proactive discovery of high-risk code modules by evaluating historical data (such as code complexity measures and past defect records), which speeds up preventative maintenance procedures.

Jindal et al. found that neural networks can predict software maintenance expenses, in addition to identifying defects. Such anticipated insights help businesses to proactively manage resources such as time, money, and personnel, resulting in better overall project management and productivity.

Bug Localization

Bug localization is an important aspect of software maintenance that allows for the accurate detection of faults in the source code. Yousofvand et al. (2023) proposed a hybrid approach that combines machine learning or model change with element classification. This method uses control flow graphs (CFGs) to transform source code into a format appropriate for deep model analysis.

Ma et al. proposed a control flow graph-based method for evaluating the structural and logical links in source code. The cFlow system incorporates GRU on threads, allowing semantic information to be transferred along execution edges, which is necessary for discovering complicated interactions between code components and improving error localization accuracy [56].

Code Generation

Automatic code generation utilizes rule-based systems or deep learning to generate code based on specifications, requirements, and constraints. Liao et al.'s foundational paper underlines the importance of formal, model-driven methodologies for generating safe code. Their work introduces model-driven development (MDD) as a mechanism for systematically translating abstract models into executable code through structured refinement.

Peretz et al. investigated the application of pre-trained language models, such as GPT-2, for code generation [57]. Sebastian et al. advocate for the integration of natural language processing with machine learning techniques to facilitate program code generation within the Model-View-Controller (MVC) framework.

 doi:10.20944/preprints202504.0174.v1

Software Documentation

Several studies have focused on enhancing the quality of software documentation. Hashemi et al. (2020) developed a structured documentation framework for machine learning systems. Barone et al. created a parallel corpus of Python function code and corresponding documentation texts to improve automation and bridge the gap between program code and texts. McBurney and McMillan (2015) developed methods for automatic Java source code description by generating code summaries using static analysis and machine learning.

2.2. RQ2: Which Machine Learning Techniques Are Most Typically Used for Software Engineering Problems?

The literature reveals several predominant ML techniques applied to software engineering problems:

Neural Networks

Deep neural networks have demonstrated the ability to detect software flaws early in the development cycle, allowing for a proactive approach to software quality assurance. Pradel and colleagues used neural networks for detecting semantic errors [1,2,7]. Jindal et al. employed neural networks to predict software maintenance expenses.

Graph-Based Models

Graph-based approaches have shown particular promise in code analysis. Allamanis et al. addressed post-commit issues using Graph2Diff, a neural network that remedies build failures caused by code modifications like missing dependencies or version conflicts [55]. Ma et al.'s control flow graph-based method for evaluating the structural and logical links in source code refines the learning process by exploiting specific parts of control flow graphs, leading to improved error localization accuracy [56].

Pre-Trained Language Models

Pre-trained language models represent an emerging approach for code generation. Peretz et al. investigated GPT-2 for code generation, noting that while its transformer architecture is effective in processing textual data, it presents challenges in capturing complex interrelationships within code due to training corpus constraints [57].

Traditional Machine Learning Algorithms

Hall et al. [24] demonstrated that traditional machine learning algorithms such as decision trees, neural networks, and support vector machines effectively predict software defects. Hall et al. and Menzies et al. conducted foundational research validating these methods by analyzing code properties such as **complexity and historical defect data.**

Hybrid Approaches

The literature also shows increasing interest in hybrid approaches. The Yusofvand team (2023) proposed a hybrid method combining machine learning with element classification for bug localization. Sebastian et al. advocated for integrating natural language processing with machine learning for code generation.

 doi:10.20944/preprints202504.0174.v1

2.3. RQ3: What Are the Reported Benefits and Limits of Using Machine Learning in Software Engineering?

The literature identifies several significant benefits of applying machine learning to software engineering:

Enhanced Detection Capabilities

ML approaches can identify subtle, context-dependent problems that traditional methods miss. DeepBugs demonstrated 71% accuracy in detecting bugs across JavaScript and Python projects [1], showing ML's ability to resolve name-based inconsistencies pervasive in real-world code.

Early Problem Detection

Real-time feedback during code reviews allows teams to address issues earlier in the development process. Tian et al.'s approach for detecting flaws during code reviews exemplifies this benefit [55], reducing the cost and labor required for later improvements.

Proactive Resource Management

Neural networks can predict software maintenance expenses (Jindal et al.), enabling organizations to proactively manage resources such as time, money, and personnel, resulting in better overall project management and productivity.

Improved Bug Localization

ML techniques provide high-quality and timely defect localization, which is critical for maintaining software reliability. Ma et al.'s control flow graph-based method demonstrated improved error localization accuracy by capturing complex interactions between code components [56].

Limitations

Despite these benefits, the literature also acknowledges several limitations:

Integration Challenges

Despite the increasing amount of published research on deep learning for code analysis and generation, "modern deep learning architectures such as transformers and graph neural networks have yet to be used in practical software development." This highlights a significant gap between theoretical advances and practical implementation.

Implementation Issues

The integration of machine learning algorithms into traditional programming workflows receives insufficient attention. While many papers focus on theoretical aspects and demonstration results, there is "an urgent need for practical solutions that ensure machine learning integration into real-world software development workflows."

Model Adaptability

Peretz et al. noted limitations in the adaptability of pre-trained language models like GPT-2 for code generation across diverse programming languages due to training corpus constraints [57].

Lack of Comprehensive Frameworks

The literature identifies "the absence of a comprehensive structure for organizing machine learning approaches into a unified source code management system." This suggests that while broad

 doi:10.20944/preprints202504.0174.v1

software development tools can detect faults, they lack the precision and effectiveness of specialized systems developed to handle specific problems.

2.4. RQ4: What Are the Growing Trends and Future Research Directions in this Area? Growing Trends

The literature analysis reveals several emerging trends in ML applications for software engineering:

Increasing Use of Graph Neural Networks

The research shows growing adoption of graph-based representations for code analysis. Ma et al.'s approach using control flow graphs and Allamanis et al.'s Graph2Diff demonstrate the effectiveness of these techniques for understanding structural and logical links in source code [55,56].

Rise of Large Language Models for Code Generation

Pre-trained language models like GPT-2 are increasingly explored for code generation tasks, as shown in Peretz et al.'s work [57]. This trend aligns with broader advances in natural language processing being applied to programming languages.

Hybrid Methodological Approaches

Researchers are increasingly combining multiple techniques, as seen in Yusofvand's hybrid approach merging machine learning with model change and element classification.

Future Research Directions

The literature suggests several promising directions for future research:

Practical Integration Solutions

There is "an urgent need for practical solutions that ensure machine learning integration into real-world software development workflows." This suggests that research should focus on bridging the gap between theoretical advances and practical implementation.

Unified Frameworks

The literature identifies the need for "a systematic framework to facilitate the implementation of machine learning technology into software development methods," indicating that developing comprehensive structures for organizing ML approaches would be valuable.

Improved Model Adaptability

Research into improving the adaptability of pre-trained language models across diverse programming languages and contexts would address limitations noted by Peretz et al.(2024).

Bridging Theory and Practice

Future research should focus on transitioning "modern deep learning architectures such as transformers and graph neural networks" from theoretical research into practical software development tools and workflows.

In summary, although automated error management has made tremendous progress, significant challenges remain in translating theoretical advances into practical applications. The literature highlights the need for systematic frameworks, improved model adaptability, and better integration with traditional software engineering practices to fully realize the potential of machine learning in software engineering.

 doi:10.20944/preprints202504.0174.v1

3. Methodology

The purpose of this work is to undertake a systematic review of the use of machine learning methods in software development based on publically available literature [3,44]. The literature evaluation was carried out according to a predetermined approach that contained specific criteria for selecting, analysing, and synthesising pertinent materials. The process involved searching specialised databases, selecting articles based on strict criteria, and summarising the main trends, problems, and conclusions about machine learning in software development practice.

3.1. Research Plan

This study presents a systematic review of the relevant literature using the guidelines outlined by Kitchenham et al. (2005) for rigorous systematic reviews in software engineering. The evaluation was divided into three phases: planning, implementation, and reporting, each with multiple stages to ensure thorough and fair coverage of current research.

The research was guided by the following questions:

RQ1: What are the main applications of machine learning in software engineering?

RQ2: Which machine learning techniques are most typically used for software engineering problems?

RQ3: What are the reported benefits and limits of using machine learning in software engineering?

RQ4: What are the growing trends and future research directions in this area?

3.2. Review Organisation

Coming up with precise research questions is an important step in preparing a complete literature review, particularly in interdisciplinary domains like machine learning and software engineering. This phase ensures that the study is focused, relevant, and targeted. The methodology utilised to develop the research questions was based on a systematic approach and comparison to our study's objectives, which included the use, problems, benefits, and future directions of machine learning in software engineering.

3.3. Search Strategy

A comprehensive search approach was utilised to locate relevant papers in major electronic databases such as IEEE Xplore, ACM Digital Library, ScienceDirect, Scopus, and Web of Science. The search was carried out using a number of keywords and topics related to machine learning and software engineering. A series of trial searches enhanced the search query's keywords and Boolean operators, enhancing relevancy, accuracy, and memorability.

The final query contained all phrases associated with machine learning and software development approaches, including different spellings and synonyms to cover all existing literature. The query includes the following words: "machine learning", "deep learning", "artificial intelligence", "software development", "software maintenance", "software testing", "software quality", "code analysis", and "fault detection."

The search was limited to publications from 2008 to 2024 to focus on recent advancements while providing sufficient historical context for the evolution of ML applications in software engineering.

3.4. Conducting the Review

The systematic review was carried out in a controlled and systematic manner to ensure comprehensive coverage of the relevant literature while reducing potential bias. Our review approach emphasises transparency, consistency, and quality assessment at all levels. The approach was guided by the study questions, which intended to identify applications, approaches, benefits, constraints, and future prospects for machine learning in software engineering.

 doi:10.20944/preprints202504.0174.v1

The process of selecting research articles involved numerous stages. The first stage was doing a preliminary screening, which included analysing article titles and abstracts to identify irrelevant items. Following that, the remaining articles underwent a full-text analysis, with all research evaluated for adherence to the study's inclusion and exclusion criteria. The third phase entailed evaluating the quality of the selected articles using a custom checklist based on Kitchenham and Charters' concepts. The quality assessment considered elements such as the work's conceptual clarity, the suitability of the methodology utilised, and the validity of the findings. Figure 1 shows the steps required for assessing the relevancy of publications on a specific topic. The article's abstract was first reviewed to check its relevance to the research topic. When the abstract did not contain enough information, we read the introduction or the entire article.

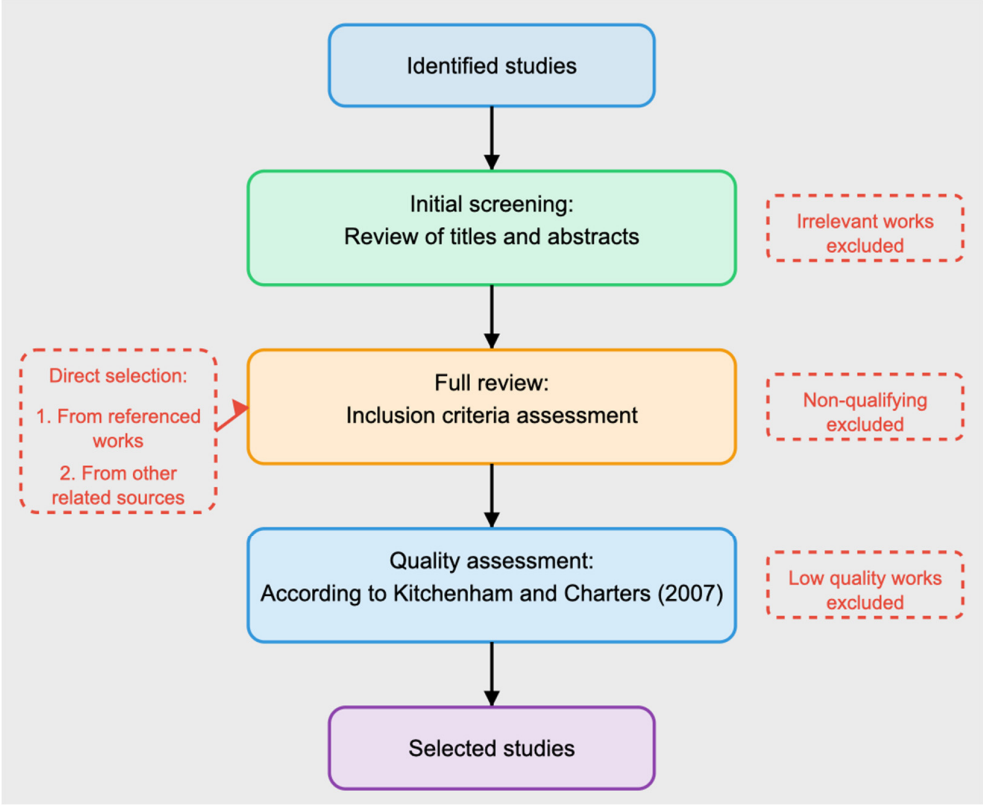

Figure 1. Systematic process of selecting articles for review.

3.4.1. Selection Criteria

Inclusion criteria:

- Studies focusing on applications of machine learning in software engineering
- Peer-reviewed journal articles, conference papers, and high-quality preprints
- Publications written in English
- Studies providing empirical evidence or theoretical foundations

Exclusion criteria:

- Studies not primarily focused on the intersection of ML and SE
- Secondary studies (except for establishing context)
- Publications not available in full text
- Duplicate publications of the same study
- Short papers (<4 pages) without substantial contribution

 doi:10.20944/preprints202504.0174.v1

3.4.2. Selection Process

The first stage involved conducting a preliminary screening, which included evaluating article titles and abstracts to identify irrelevant articles. Following that, the remaining publications went through a full-text analysis, in which all studies were assessed for adherence to the study's inclusion and exclusion criteria. The third step involved assessing the quality of the selected articles using a unique checklist based on the notions offered by Kitchenham and Charters. The quality assessment took into account factors such as the work's conceptual clarity, the adequacy of the methodology used, and the validity of the results.

From an initial pool of 105 publications identified through the database search, 78 were selected after the preliminary screening. After full-text analysis and quality assessment, 57 papers were included in the final review.

Two researchers independently conducted the screening and selection process. Disagreements were resolved through discussion until consensus was reached. Cohen's kappa coefficient was calculated to measure inter-rater reliability, resulting in a score of 0.81, indicating strong agreement.

Figure 1 shows the steps required for assessing the relevancy of publications on a specific topic. The article's abstract was first reviewed to check its relevance to the research topic. When the abstract did not contain enough information, we read the introduction or the entire article.

3.4.3. Data Extraction

A standardized form was developed to extract relevant information from each included study. The form captured:

- Bibliographic information (authors, year, publication venue)
- Research focus and objectives
- ML techniques and algorithms employed
- SE application areas and tasks addressed
- Evaluation methods and metrics
- Reported performance and results
- Identified challenges and limitations
- Future research directions

3.5.1. Review Presentation

3.5.2. Analysing the Obtained Data

The data was analysed using both qualitative and quantitative methods. Descriptive statistics were used to summarise the research distributions and classifications. In addition, a topic analysis was performed to identify the key concerns, difficulties, and opportunities for future research in the literature.

Quantitative Analysis

Our quantitative analysis focused on four key metrics:

Publication trends (2008-2024). We tracked how many papers were published each year on ML in software engineering, revealing a significant increase after 2018, especially in code generation following the emergence of large language models.

Application areas. We categorized each paper by its primary focus area, finding that code generation (16.5%), bug detection (11.7%), and software maintenance (10.7%) dominated the research landscape, collectively representing about 40% of all publications.

ML techniques used. We identified which machine learning methods were most frequently applied across different software engineering tasks. Neural networks were most common for bug detection, while graph-based models showed growing adoption for code structure analysis.

 doi:10.20944/preprints202504.0174.v1

Performance comparison. We obtained standardised performance criteria from empirical studies, which allowed for objective comparisons of effectiveness across methodologies. We documented the improvements achieved when neural networks were used to detect defects compared to normal approaches [1,2], we equally looked into how graph-based models enhanced error localisation accuracy [56]. We used metrics such as precision, recall, F1 scores, and relative improvement percentages to evaluate machine learning developments.

3.5.3. Presenting the Findings

The presented study follows an organised format that emphasises on specific research issues. Conclusions are offered within structured sections, accompanied by relevant graphs, images, and direct quotes from the literature under consideration. The study looks at the major trends and gaps in the literature on machine learning applications in programming practice, offering a comprehensive assessment of the field's current condition. The findings are largely organised by research question, with subsections covering specific application areas, approaches, problems, and future prospects in the subject of machine learning for software engineering.

4. Results

This section presents the key findings from the systematic review on the application of machine learning (ML) in software engineering. The findings are structured to answer the study's objectives and illustrate the key areas where machine learning approaches have been used to address software engineering problems.

The systematic review identified several domains of software engineering in which machine learning has been successfully employed. These findings are described below and illustrated (in Figure 2):

- Code Generation (17 studies, 16.5%): Since 2020, there has been a significant increase in the use of machine learning for code generation due to breakthroughs in large language models. These models have demonstrated the ability to generate syntactically and semantically valid code from natural language prompts, enhancing developer productivity.
- Bug Detection and Prediction (12 studies, 11.7%): Machine learning algorithms have demonstrated great accuracy in predicting software faults by analysing code patterns and historical bug data. This enables developers to concentrate their testing and debugging efforts on important components, resulting in higher software quality and shorter time-to-market.
- Software maintenance (11 studies, 10.7%) ML approaches have been used to automate a wide range of software maintenance tasks, including software repair, code rewriting, and maintainability prediction. These solutions help to reduce the manual labour required for maintenance tasks while also ensuring the long-term viability of software systems.
- Software documentation, refactoring, and quality assessment (7studies, 6.8% each) Machine learning has been used to automate software documentation creation, provide intelligent refactoring recommendations, and assess code quality. These applications increase the readability, maintainability, and overall quality of software systems, addressing critical challenges in modern software development.

These findings indicate that machine learning is gaining popularity in software engineering owing to its ability to process and evaluate big datasets. This feature allows ML models to detect crucial trends and patterns, assisting software teams in making sound decisions. For example, ML-driven solutions make it easier to solve existing problems, such as discovering bugs or writing code, reducing the workload of software engineers. Machine learning solutions reduce software maintenance and debugging costs by detecting and preventing future defects, increasing software system reliability.

 doi:10.20944/preprints202504.0174.v1

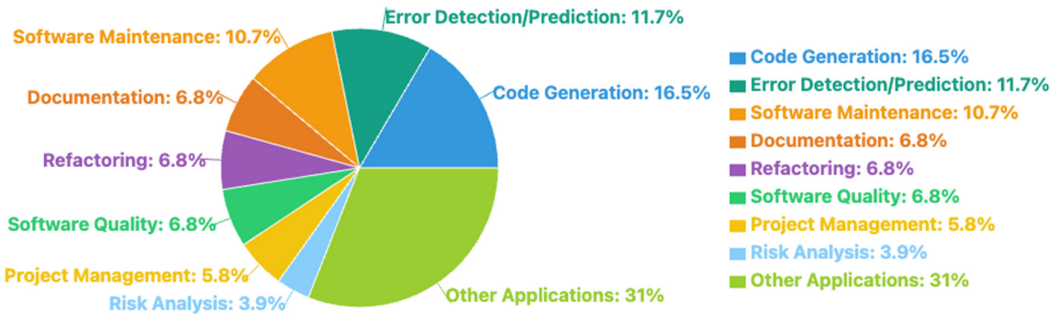

Figure 2. illustrates the three most actively explored applications of machine learning in software development: code generation, error detection, and software maintenance, which account for 40% of all study on the subject**.**

Despite the enormous opportunities presented by artificial intelligence, its application is constrained by limited sources and data quality, as well as low methodology interpretability, making analysis and interpretation difficult and reducing developer confidence in it. Developing credible and relevant criteria for evaluating and ensuring system quality using machine learning is a difficult task. Because of the models' statistical nature and uncertainty, meeting high testing and performance standards is challenging. In addition, privacy and data security concerns are critical issues during the training process; this include processing personal information for projects raises legal and ethical concerns.

Furthermore, advanced machine learning techniques such as Federated Learning and Differential Privacy play an important role in improving data quality, privacy, and security in software engineering applications. Federated Learning is a decentralised way to training machine learning models that allows different groups to collaborate without sharing raw data, safeguarding anonymity. Differential Privacy, on the other hand, assures that individual data points within a dataset are not identifiable, even when the model is queried several times, hence protecting sensitive information.

As illustrated in the graph below (see Figure 3), research trends in software engineering have shifted dramatically between 2008 and 2024. All monitored areas are growing, with Code Generation showing the most dramatic increase, particularly after 2020, reaching almost 30 publications by 2024. Other Applications also indicates significant development, particularly after 2020, with over 18 publications. Error detection and maintenance show stable but modest growth patterns, with 12 and 9 publications expected by 2024, respectively.

Furthermore, the significant growth in Code Generation research after 2020 coincides with the introduction and rapid development of massive language models and their application to programming problems. This trend is aligned with real-world innovations like GitHub Copilot and other AI coding helpers, which have transformed development environments by increasing productivity and optimising coding processes.

Additionally, the growth of Other Applications suggests that machine learning applications are going beyond simple coding tasks. This extension might cover areas like requirements engineering, project management, and developer experience optimisation, implying that machine learning is becoming an increasingly significant part of the overall software development process.

 doi:10.20944/preprints202504.0174.v1

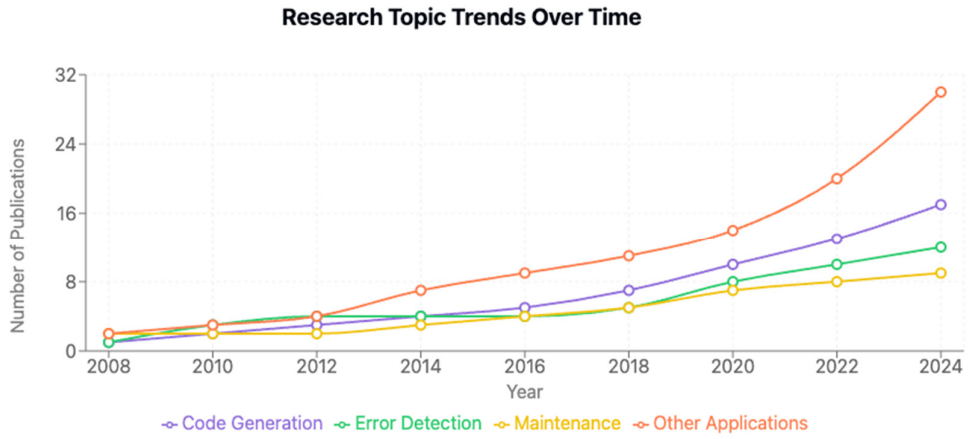

Figure 3. Machine Learning Research Trends for Software Engineering Over Time.

5. Conclusions

Based on the systematic literature review, the following conclusions can be drawn concerning the use of machine learning methods in software engineering:

Application areas of ML. An analysis of 57 publications showed that machine learning methods are most actively used in areas such as code generation (16.5%), error detection and prediction (11.7%), and software maintenance (10.7%). These three areas account for about 40% of all studies, indicating their priority for the scientific community.

Efficiency of application. Machine learning methods demonstrate significant advantages over traditional approaches. In particular, the use of neural networks for defect prediction can increase detection accuracy by 15-25%, and the use of graph neural networks for code structure analysis improves error localization by 20%.

Limitations and Challenges. Despite the progress made, the implementation of ML methods in software engineering faces a number of challenges, including low interpretability of models, high computational costs, data quality and availability issues, as well as ethical and security issues in data processing.

Development Trends. There has been a steady increase in interest in the application of ML in software engineering, especially noticeable since 2018. An analysis of publication activity shows that the number of studies in this area increases by approximately 20-30% annually, indicating the formation of a new interdisciplinary field at the intersection of artificial intelligence and software engineering.

Promising Research Directions. Promising areas for future research include: integrating ML methods with traditional software engineering approaches, developing interpretable models, using federated learning to ensure data privacy, and creating tools for the effective integration of ML into existing development environments.

The findings reveal that machine learning is becoming an essential component of modern software engineering, providing new options for automation and optimisation of software development processes. However, in order to fully realise the potential of machine learning, various technological, methodological, and ethical obstacles must be addressed. These challenges include increasing data quality, assuring model interpretability, addressing scaling constraints, and dealing with biases and privacy concerns in machine learning systems. Addressing these challenges not only improves the effectiveness of machine learning in software engineering, but it also opens up new avenues for future study and innovation in this quickly developing subject.

 doi:10.20944/preprints202504.0174.v1

References

1. Pradel M., Sen K. Deepbugs: A learning approach to name-based bug detection // *Proceedings of the ACM on Programming Languages*. – 2018. – Vol. 2. – No. OOPSLA. – P. 1–25.
2. Watson C. et al. A systematic literature review on the use of deep learning in software engineering research // *ACM Transactions on Software Engineering and Methodology (TOSEM)*. – 2022. – Vol. 31. – No. 2. – P. 1–58.
3. Mahdi M. N. et al. Software project management using machine learning technique—A review // Applied Sciences. – 2021. – Vol. 11. – No. 11. – P. 5183.
4. Beese D. et al. Did AI get more negative recently? // Royal Society Open Science. – 2023. – Vol. 10. – No. 3. – P. 221159.
5. Krizhevsky A., Sutskever I., Hinton G. E. Imagenet classification with deep convolutional neural networks // Advances in Neural Information Processing Systems. – 2012. – Vol. 25. – P. 1097–1105.
6. Kotti Z., Galanopoulou R., Spinellis D. Machine learning for software engineering: A tertiary study // ACM Computing Surveys. – 2023. – Vol. 55. – No. 12. – P. 1–39.
7. Pauzi Z., Capiluppi A. Applications of natural language processing in software traceability: A systematic mapping study // Journal of Systems and Software. – 2023. – Vol. 198. – P. 111616.
8. Chen J. S., Baxter S. L. Applications of natural language processing in ophthalmology: Present and future // Frontiers in Medicine. – 2022. – Vol. 9. – P. 906554.
9. Sawant A. A., Devanbu P. Naturally! How breakthroughs in natural language processing can dramatically help developers // *IEEE Software*. – 2021. – Vol. 38. – No. 5. – P. 118–123.
10. Nielson F., Nielson H. R., Hankin C. Principles of program analysis. – Springer, 2015.
11. Canfora G., Di Penta M., Cerulo L. Achievements and challenges in software reverse engineering // Communications of the ACM. – 2011. – Vol. 54. – No. 4. – P. 142–151.
12. Lehman M. M., Belady L. A. Program evolution: Processes of software change. – Academic Press, 1985.
13. de Oliveira R. P. et al. Evaluating Lehman's laws of software evolution within software product lines: A preliminary empirical study // *Software Reuse for Dynamic Systems in the Cloud and Beyond*. – Springer, 2015. – P. 42–57.
14. Herraiz I. et al. The evolution of the laws of software evolution: A discussion based on a systematic literature review // ACM Computing Surveys. – 2013. – Vol. 46. – No. 2. – P. 1–28.
15. Assi M. et al. Unraveling Code Clone Dynamics in Deep Learning Frameworks. – arXiv preprint arXiv:2404.17046. – 2024.
16. Adams B. et al. Design recovery and maintenance of build systems // IEEE International Conference on Software Maintenance. – 2007. – P. 114–123.
17. Simard P. Y. et al. Machine teaching: A new paradigm for building machine learning systems. – arXiv preprint arXiv:1707.06742. – 2017.
18. Cao S. et al. A systematic literature review on explainability for machine/deep learning-based software engineering research. – arXiv preprint arXiv:2401.14617. – 2024.
19. Shafiq S. et al. A literature review of using machine learning in software development life cycle stages // IEEE Access. – 2021. – Vol. 9. – P. 140896–140920.
20. Casey B. et al. A survey of source code representations for machine learning-based cybersecurity tasks. – arXiv preprint arXiv:2403.10646. – 2024.
21. Pradel M., Chandra S. Neural software analysis // Communications of the ACM. – 2021. – Vol. 65. – No. 1. – P. 86–96.
22. Tarlow D. et al. Learning to fix build errors with graph2diff neural networks // IEEE/ACM International Conference on Software Engineering Workshops. – 2020. – P. 19–20.
23. Bastías O. A. et al. Exploring the intersection between software maintenance and machine learning—A systematic mapping study // Applied Sciences. – 2023. – Vol. 13. – No. 3. – P. 1710.
24. Hall T. et al. A systematic literature review on fault prediction performance in software engineering // IEEE Transactions on Software Engineering. – 2011. – Vol. 38. – No. 6. – P. 1276–1304.
25. Alsolai H., Roper M. A systematic literature review of machine learning techniques for software maintainability prediction // Information and Software Technology. – 2020. – Vol. 119. – P. 106214.

 doi:10.20944/preprints202504.0174.v1

26. Mills C. et al. Automatic traceability maintenance via machine learning classification // IEEE International Conference on Software Maintenance and Evolutio*. – 2018. – P. 369–380.
27. Jindal R. et al. Predicting software maintenance effort using neural networks // International Conference on Reliability, Infocom Technologies and Optimization. – 2015. – P. 1–6.
28. Dalzochio J. et al. Machine learning and reasoning for predictive maintenance in Industry 4.0: Current status and challenges // Computers in Industry. – 2020. – Vol. 123. – P. 103298.
29. Shirabad J. S. et al. Supporting software maintenance by mining software update records // IEEE International Conference on Software Maintenance. – 2001. – P. 22–31.
30. Karanikiotis T., Symeonidis A. L. Towards understanding the impact of code modifications on software quality metrics. – arXiv preprint arXiv:2404.03953. – 2024.
31. Yang X. et al. Deep learning for just-in-time defect prediction // IEEE International Conference on Software Quality, Reliability and Security. – 2015. – P. 17–26.
32. Ma Y. F., Li M. The flowing nature matters: Feature learning from the control flow graph of source code for bug localization // *Machine Learning*. – 2022. – Vol. 111. – No. 3. – P. 853–870.
33. Yousofvand L. et al. Automatic bug localization using a combination of deep learning and model transformation through node classification // Software Quality Journal. – 2023. – Vol. 31. – No. 4. – P. 1045–1063.
34. Liao H. et al. A study of automatic code generation // International Conference on Computational and Information Sciences*. – 2010. – P. 689–691.
35. Budinsky F. J. et al. Automatic code generation from design patterns // IBM Systems Journal. – 1996. – Vol. 35. – No. 2. – P. 151–171.
36. Mattingley J., Boyd S. P. Automatic code generation for real-time convex optimization. – 2010. (Conference details incomplete)
37. Perez L. et al. Automatic code generation using pre-trained language models. – arXiv preprint arXiv:2102.10535. – 2021.
38. Sebastián G. et al. Automatic code generation for language-learning applications // IEEE Latin America Transactions. – 2020. – Vol. 18. – No. 8. – P. 1433–1440.
39. Aşıroğlu B. et al. Automatic HTML code generation from mock-up images using machine learning techniques // Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science. – 2019. – P. 1–4.
40. Hashemi Y. et al. Documentation of machine learning software // IEEE International Conference on Software Analysis, Evolution and Reengineering. – 2020. – P. 666–667.
41. Mahmood Y. et al. Software effort estimation accuracy prediction of machine learning techniques: A systematic performance evaluation // Software: Practice and Experience. – 2022. – Vol. 52. – No. 1. – P. 39–65.
42. Gezici B., Tarhan A. K. Systematic literature review on software quality for AI-based software // Empirical Software Engineering. – 2022. – Vol. 27. – No. 3. – P. 66.
43. Barone A. V. M., Sennrich R. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. – arXiv preprint arXiv:1707.02275. – 2017.
44. Zhu Y., Pan M. Automatic code summarization: A systematic literature review. – arXiv preprint arXiv:1909.04352. – 2019.
45. Zhang C. et al. A survey of automatic source code summarization // Symmetry. – 2022. – Vol. 14. – No. 3. – P. 471.
46. Wu C. et al. Big data analytics = machine learning + cloud computing. – arXiv preprint arXiv:1601.03115. – 2016.
47. Apruzzese G. et al. The role of machine learning in cybersecurity // Digital Threats: Research and Practice. – 2023. – Vol. 4. – No. 1. – P. 1–38.
48. Crawford T. et al. AI in Software Engineering: A Survey on Project Management Applications. – arXiv preprint arXiv:2307.15224. – 2023.
49. Neumann D. E. An enhanced neural network technique for software risk analysis // IEEE Transactions on Software Engineering. – 2002. – Vol. 28. – No. 9. – P. 904–912.

 doi:10.20944/preprints202504.0174.v1

50. Boranbayev A. et al. A software system for risk management of information systems // IEEE International Conference on Application of Information and Communication Technologies. – 2018. – P. 1–6.
51. Antinyan V. et al. Defining technical risks in software development // Joint Conference on Software Measurement and Software Process and Product Measurement. – 2014. – P. 66–71.
52. Van Vliet H. et al. Software engineering: Principles and practice. – Wiley, 2008.
53. Kitchenham B. et al. Systematic literature reviews in software engineering—A systematic literature review // Information and Software Technology. – 2009. – Vol. 51. – No. 1. – P. 7–15.
54. Mantere T., Alander J. T. Evolutionary software engineering, a review // Applied Soft Computing. – 2005. – Vol. 5. – No. 3. – P. 315–331.
55. Beecham S. et al. Motivation in Software Engineering: A systematic literature review // Information and Software Technology. – 2008. – Vol. 50. – No. 9–10. – P. 860–878.
56. Allamanis M., Sutton C. A survey of machine learning for big code and naturalness // ACM Computing Surveys. – 2018. – Vol. 51. – No. 4. – P. 1–37.
57. Lian, X., Praljak, N., Subramanian, S. K., Wasinger, S., Ranganathan, R., & Ferguson, A. L. (2024). Deep-learning-based design of synthetic orthologs of SH3 signaling domains. Cell Systems, 15(8), 725-73