


Research Article

An Automated Tool for Streamlining Software Engineering: Information Extraction and Decision

¹Rasha Gh. Alsarraj 

Software

Computer Science and Mathematics

University of Mosul, Mosul, Iraq

rasha.alsarraj@uomosul.edu.iq

ARTICLE INFO

Article History

Received: 05/02/2025

Accepted: 03/05/2025

Published: 25/08/2025

This is an open-access article under the CC BY 4.0 license:

<http://creativecommons.org/licenses/by/4.0/>



ABSTRACT

In the always-evolving and dynamic field of software development, good decision-making is absolutely critical. Developers have to regularly decide how best to apply features, optimize performance and debug issues. This process could be much improved by extracting actionable insights from software code. The presented work explores the tools and metrics available to enable developers to make data-driven decisions, therefore enhancing the development efficiency as well as code quality. Also, it introduces a new automated tool called CodeLens which analyzes software code, extract lines of code (LOC), documentation quality, complexity, and other key criteria. Through a consolidated view of such metrics, the tool helps developers evaluate code fit, spot possible bottlenecks, and prioritize optimization or refactoring efforts. Furthermore, the tool's support of Java and Python languages guarantees general applicability, hence fitting for many software projects.

Keywords: *Developer Decision Support; Code Analysis; Software Assessment; Software Quality; and Software Metrics.*

1. INTRODUCTION

In the fast-expanding field of software engineering, developers sometimes have to make difficult decisions under tight deadlines. Those decisions affect how performance, maintainability, and quality regarding software systems remain [1]. Still, it is quite difficult to derive voluminous and complex codebases into actionable insights. Meaningful patterns and metrics extracted from software code provide a means of data-driven, informed decision-making. Developers have quantifiable data from metrics, like test coverage, code complexity, security vulnerabilities, and performance bottlenecks which could direct their decision-making [2]. By helping to identify areas needing refactoring, optimization, or extra testing, such data-driven method guarantees a more consistent and high-performance result. Studies have found, for example, that code complexity closely corresponds with maintenance difficulties and that high complexity is usually sign of bugs or performance problems. Objective measure regarding code fitness could be obtained by means of tools and approaches assessing such criteria: LOC, code complexity, function/method analysis, and code duplication [3]. Furthermore, including such criteria into the software development lifecycle—particularly in the phases of testing, development, and maintenance—helps to improve decision-making. Teams may guarantee that quality is kept throughout the development process instead of at the end through constantly monitoring such measures using automated technologies linked with Continuous Deployment (CD) and Continuous Integration (CI) pipelines [4]. Instead of waiting for later-stage discovery of possible problems, the real-time feedback such technologies provide helps developers to constantly modify their strategies. Even with the

possible advantages, though, obtaining and analyzing metrics from large codebases comes with difficulties. Manual examination is time-consuming and error-prone in codes since they sometimes show complex relations, modular structures, and several layers of abstraction [5]. Streamlining such procedure depends much on automation using software tools, which provides code quality analysis and detects problems in many aspects. Together with appropriate metric selection, such tools provide a clearer view regarding the condition of the software, which helps developers make better decisions directly affecting the general product quality [6]. This article presents a new tool meant to help developers make informed decisions all across the software development lifecycle by parsing software code. Through offering analysis of important indicators such documentation quality, code complexity, and resource efficiency, the suggested solution helps developers to maximize their processes and provide better-quality software. The tool's capacity to handle Java and Python codebases guarantees its relevance over many different projects, Since these two languages are widely used currently. This study mostly offers the following contributions:

- 1- Development of a Code Parsing Tool: The suggested tool extracts from software code important data including complexity, lines of code, duplication ratio, and memory use, such easily available metrics help developers to evaluate code complexity and condition.
- 2- Enhanced Decision-Making Support: Consolidating important insights into a single platform helps developers to find bottlenecks, prioritize refactoring projects, and guarantee improved resource allocation.
- 3- Applicability across Languages: Supporting Java and Python, the tool fits a broad range of software development environments, therefore enhancing its adaptability and possible adoption.
- 4- Facilitation of Best Practices: Automating the presentation and extraction of software metrics helps the tool support adherence to best practices in testing, coding, and documentation.

The related work in section 2, the metrics obtained by the tool explained in section 3, the approaches utilized for building tool as well as the ways where such insights could drive in section 4, the case study present in section 5, after that discussion and results explained in section 6, then a comparison with other tools present in section 7, lastly the conclusion present in section 8.

2. RELATED WORKS

Recent studies in software engineering focusing on code analysis, quality assurance, and maintenance offer critical insights into the challenges and advancements within this domain, which depending on information extracted from code.

In 2019, Mamun et al. conducted a comprehensive study to investigate the effects of measurement techniques on the correlations of software code metrics. They highlighted the importance of methodological rigor in empirical software engineering, emphasizing how different measurement choices can significantly influence the observed relationships between metrics. The authors found inconsistencies in previously published metric correlations through systematically analyzing several datasets as well as measurement techniques, therefore highlighting the possibility of misleading conclusions in the case when methodological factors are neglected. They gave practitioners and researchers instructions to guarantee the validity of metric-based studies, therefore helping to improve empirical approaches in software engineering [7].

In 2020, Zagane et al. used software code metrics as features and investigated the application of deep learning (DL) approaches for software vulnerability detection. Their study emphasizes how well deep neural networks (DNNs) combine with automatically generated code metrics to efficiently predict vulnerabilities. The work shows how DL could find patterns in code that conventional methods could overlook through combining advanced learning models with static code properties. Under the framework of vulnerability prediction, the authors underlined the importance of feature extraction as well as representation and the use of software measurements as inputs for training neural networks (NNs) [8].

In 2021, Jiang et al. published Cure, a code-aware neural machine translation (NMT) model meant for automatic program repair (APR). They underlined the limits of conventional NMT-based APR methods, which may ignore syntactic and structural constraints particular to programming languages. Through introducing code-aware elements including Abstract Syntax Tree (AST) representations and tokenized code inputs to improve the model's

understanding regarding program structure, Cure addresses such challenges. Leveraging both contextual knowledge and semantic correctness, they made notable increases in repair accuracy [9].

In 2022, Duque-Torres et al. addressed a fundamental difficulty in metamorphic testing (MT) by investigating with the use of source code metrics for the prediction of the metamorphic relations (MRs) at the method level. They looked at how predictors for MRs—which are crucial for evaluating programs without an oracle—static code properties including coupling, complexity, and cohesion might be found. The work shows the viability of using code metrics to automate and improve MR identification by use of an extensive dataset of code and application of machine learning (ML) methods [10].

In 2023, Park et al. suggested a technique to visualize software quality by use of normalizing static code building information. They sought to organize data for better comparability so that complex quality metrics could be interpreted more simply. The framework improved knowledge of software properties like complexity and maintainability. Their efforts expanded on earlier studies in software visualization and static analysis, offering tools to help with quality assessment decision-making. The authors provide a useful approach for enabling developers to make software quality measurements more actionable through combining table normalization with visualization methods. Their approach sought to raise awareness of and application for quality indicators in software development projects [11].

In 2024, Mashhadi et al. projected bug frequency by means of static analysis as well as source code metrics. Their research included measures like code complexity and churn to enhance earlier work on defect prediction. They showed how this analysis might help to estimate severity more precisely, therefore offering a better basis for bug effect prediction. They developed on earlier research using statistical approaches and ML to predict defects. The authors demonstrated how valuable it is to combine analysis methods with static code features. Their results underlined the need of better models in the prediction of software quality. By more consistent severity assessments, they helped to improve software defect management [12]. In the same year, Huang et al. enhanced comment generating models by means of bytecode for extracting more semantic information from code. As a low-level, machine-readable representation, bytecode allowed the model to record information that higher level code analysis would overlook. The method improved program behavior by means of control flow graphs as well as transformer-based NNs. Emphasizing the need of obtaining more detailed knowledge from code, their approach greatly raised the accuracy and relevance of produced comments. Their use of bytecode for improved semantic extraction advanced automated software documentation. They demonstrated how more accurate software documentation could result from lower-level code representations enhancing comment [13].

In 2025, Jiang et al. enhanced code summarization prompt tuning by means of meta-data derived from codes. To direct the model in producing more accurate summaries, their approach included more meta-information including variable names as well as function signatures. Using such details will help the model for extracting more semantic insights, thereby increasing the relevance regarding the produced summaries. Emphasizing the need of information extraction from code, the study focused on how meta-data clarifies context and program behavior for the model. Building on earlier code summarizing studies, this study shows that meta-data greatly enhances the quality of summary. The method underlined the need of more accurate code interpretation depending on richer, context-aware information. Their results offer a new path for improving automated code summarization tools [14].

By means of a comprehensive, automated solution which extracts important software information and metrics like documentation quality, code complexity, dead code detection, and memory consumption, the suggested CodeLens tool presents a major breakthrough over current software analysis tools. CodeLens distinguishes itself from other tools that concentrate on particular elements or support limited programming languages through providing broad application, supporting both Java and Python, which qualifies for a wide range of software projects. Moreover, its real-time feedback features and simple interface let programmers quickly evaluate code quality and guide decisions all through the software development life. Different from other solutions, such special combination of features puts CodeLens as a potent instrument in the continuous endeavor to enhance decision-making in software development.

3. SOFTWARE INFORMATION EXTRACTION

Modern software engineering methods depend much on the extracted information from software codes. Scholars and organizations could better understand the behavior, structure, and evolution of a system through systematically examining and deriving significant insights from code [15]. This technique helps to find code smells, uncover hidden dependencies, and spot patterns influencing maintainability and software quality. Such extracted information is important since it can guide decision-making made at several phases of the software development life [16]. For performance improvement, debugging, and coding standard compliance, for example, both dynamic and static code analysis approaches offer essential data. Comparably, mining code repositories track historical trends to support predictive maintenance and better resource allocation [17][18]. Furthermore, the extracted insights helps to solve new issues in the software field including controlling technical debt, enabling automated refactoring, and guaranteeing code consistency in large-scale systems[19]. Machine learning (ML) as well as data-driven techniques have greatly opened the possibility to use acquired code information, thereby supporting intelligent recommendation systems as well as predictive analytics. Basically, the information extraction and use from software code enable developers, maintainers, and researchers to improve software reliability, adaptability, and scalability, thus guaranteeing long-term sustainability in an always changing technological landscape [20][21]. Figure 1 shows a three-main phase cycle of constant improvement for software split into planning, decision-making, and information extraction. The process starts in the planning phase, in which case aims, problem-solving, and criteria-establishing define the extent. Procedures defined in this phase define standards, techniques for data collecting, analysis, and feedback systems. The information extraction phase follows, in which data is acquired by methods of recording and storage together with procedures of review and improvement. The collected data then is examined to create reports and show results. The last phase, decision-making, emphasizes on improving the process by assessing results and development, thereby facilitating well-informed choices that advance it even more. This cycle stresses an iterative approach meant to enhance procedures constantly and reachable, meaningful improvements.

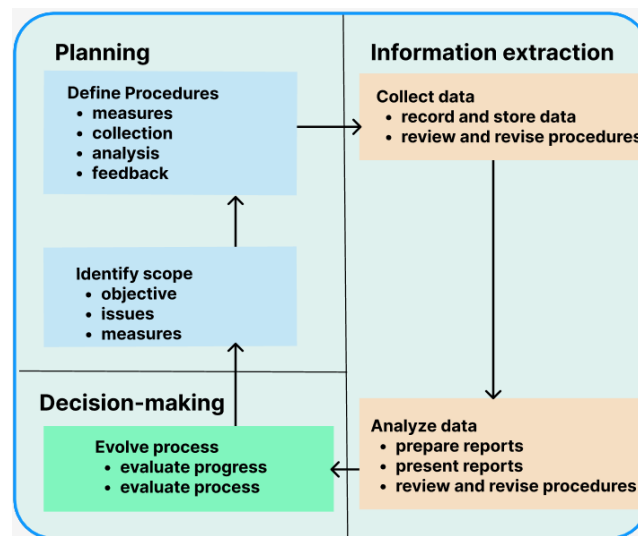


Fig. 1. Software improvement cycle

Extracted information from software code is critical for computing various software metrics, which are quantitative measures that provide insight into the quality, and performance of a codebase [22]. Extracting these metrics allows developers to identify improvement areas and make data-driven decisions. Software measurement is a progressing cycle where information on the cycle of product improvement and its items are recognized, gathered and examined for the purpose of comprehending and screening the cycle and its items and to give valuable data for the purpose of improving the cycle and its products [23]. Without estimation, developer can't make quality programming, for accomplishing fundamental administration goals of the assessment, improvement, and cycle changes, estimation is necessary[24]. The properties of good measurements include:

- **Reliability:** The yield of estimation cycle should be accurate. In addition to that, comparative aftereffects of the estimation cycle extra time and across the circumstances [25][26].
- **Sensitivity:** in a case where there is an event or trigger, the estimating component uncovers the changeability in the reactions [27][28].

- Validity: The measurement process tests what it professes to calculate [29][30].

4. METHODOLOGY

The methodology outlines the systematic approach employed to design, develop, and validate the proposed automated tool (CodeLens) for software information extraction. This process began with identifying key metrics critical for assessing software quality and its importance as illustrate in table 1 including:

- 1- Lines of Code (LOC): Total number of lines in the code, including comments and blank lines.
- 2- Comment Lines: Number of lines containing comments in the code.
- 3- Function Count: Total number of functions or methods in the code.
- 4- Code Complexity: Number of independent execution paths in the code.
- 5- Documentation Ratio: Ratio of comment lines to total lines of code.
- 6- Memory Usage: Memory consumed by the code during execution.
- 7- Nesting Depth: Maximum depth of nested structures like loops and conditionals.
- 8- Duplication Ratio: Percentage of duplicate code in the project.
- 9- Class Count: Number of classes defined in the codebase.
- 10- Dead Code Detection: Identifies code that is not referenced or executed during program execution.

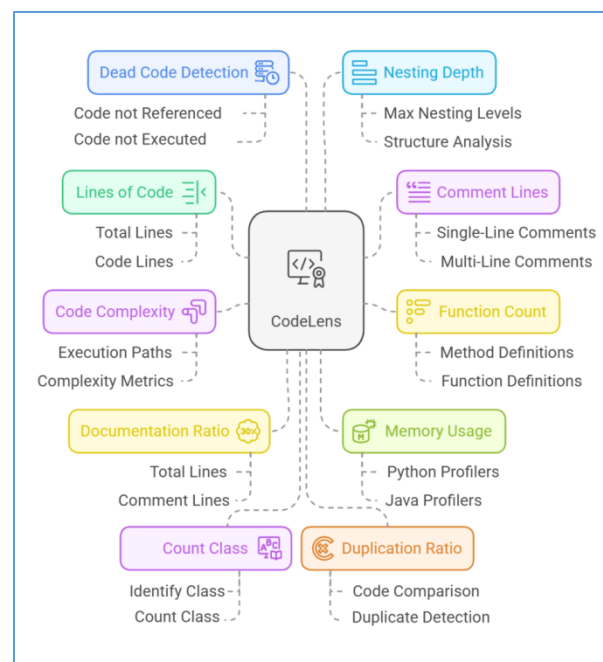


Fig. 2. CodeLens methodology

The tool steps to analysis and parse code illustrate in the Figure 2. After building the code of tool, the interface integrated with the code to make the tool more simple and visual to the user which contain the Browse and Extract Information button, in addition to eleven edit box one that show path of chosen file and the other ten for show result of extracted information as shown in figure (3) and (4). When clicked buttons including:

- Browse: Opens a file dialog to select a file and displays the file path in the first edit box.
- Extract Information: Simulates analysis of the chosen file and displays the result in the others edit boxes.

Table I. Code metrics covered by CodeLens tool.

No.	Metric	Type	Importance	Implementation Phase	Benefit From It
1.	Lines of Code (LOC)	Code Size	Measures the size of the code and reflects its complexity.	Development, Maintenance	Estimate the effort required for maintenance and development. Estimate project cost.
2.	Lines of Comments	Documentation	Measures the extent of code documentation.	Development, Maintenance	Improve documentation quality and reduce the time required to understand the code. A high percentage of comments reflects quality documentation.
3.	Number of Functions	Code Modularity	Measures the number of functions or methods in the code.	Development, Maintenance	Determine the reusability and maintainability of the code. A large number of functions may indicate excessive modularization.
4.	Code Complexity	Complexity	Measures the number of execution paths in the code.	Development, Maintenance	Identify the complex parts of the code that need refactoring. High complexity means difficulty in testing and maintenance.
5.	Documentation Ratio	Documentation	Measures the ratio of comments to total lines of code.	Development, Maintenance	Improve documentation quality and reduce the time required to understand the code. A low ratio reflects poor documentation.
6.	Memory Usage	Performance	Measures the amount of memory used during code execution.	Testing, Maintenance	Improve code performance and reduce resource usage. High memory usage may indicate performance issues.
7.	Nested Depth	Complexity	Measures the level of nesting in the code.	Development, Maintenance	Identify the complexity of the code and difficulty in understanding it. High nesting depth means difficulty in maintenance.
8.	Duplicate Code Ratio	Reusability	Measures the percentage of duplicated code in the project.	Development, Maintenance	Improve code reusability and reduce its size. Duplicated code increases maintenance difficulty.
9.	Number of Classes	Code Structure	Measures the number of classes in the code.	Development, Maintenance	Understand the structure of the code and its complexity. A large number of classes may indicate excessive complexity.
10.	Dead Code Detection	Optimization	Detect the dead code	Testing, Maintenance	makes the codebase cleaner and easier to understand. Optimizes memory usage by removing unused entities.

5. Case study

The case study showcases the practical use of the Codelens tool in real-world scenarios, demonstrating its versatility across various software projects. In this study, the Codelens tool is applied to two popular open-source software repositories obtained from GitHub: Flask (a Python web framework) and JUnit (a widely used testing framework in Java). These repositories were chosen due to their widespread usage and complexity, providing a robust testing ground for the CodeLens tool. The tool has two basic steps including:

- **Browsing the Code in CodeLens:** Following software file downloads, they have been imported into the CodeLens tool for analysis. The tool was set to search the code, compile important statistics on the codebase like Lines of Comment (LC), Lines of Code (LOC), Code Complexity (CC), and more.
- **Extracting Information:** The tool's central ability was applied for extracting from the code comprehensive metrics. Clicking the "Extract Information" button allowed the tool to process the code and show the findings in an easily interpretable format, therefore enabling the discovery of important elements including unused functions, documentation coverage, and code duplication.

Figure (3) and (4) shows the result of flask and junit software respectively.

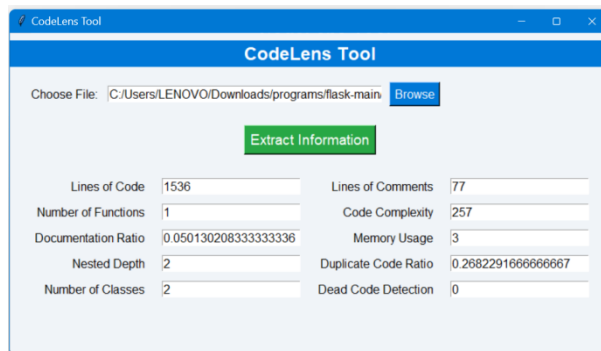


Fig. 3. Extracted Information of flask software

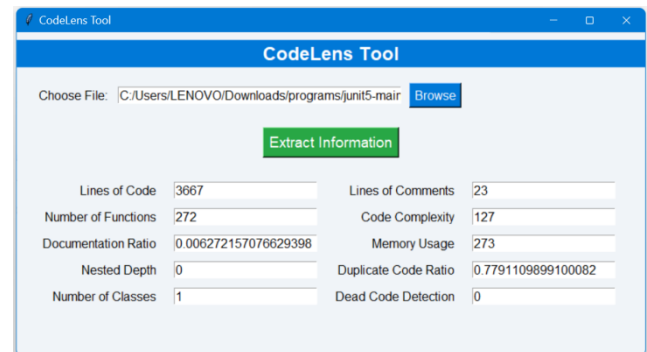


Fig. 4. Extracted Information of junit software

6. Results and Discussion

For assessing the overall quality regarding software projects, the Codelens tool turned out to be a useful tool. Easy identification of possible problems, like unused code, code duplication, and too complex code was made possible by the capacity for extracting important metrics and present them in a complete form. Applying this tool to real-worlds projects like JUnit and Flask shows clearly that it could be used in a broad range of software development environments to guarantee code quality and maintainability. Using color gradients, Figure (5) shows the degree of connection between certain code metrics whereby the colors depict different degrees of interaction between these metrics. With brighter colors denoting a more important association, the heatmap shows the degree of the relations between the measures. Lighter colors suggest less connection. Those relations could provide insightful analysis of how various metrics interact and how changes to one metric might affect other measures. The relations among the collected data including:

- Lines of Code (LOC) and Lines of Comments (LC):**
 There is typically a positive correlation between the number of lines of code (LOC) and the number of lines of comments (LC). In well-written code, it is essential to maintain an appropriate balance of comments alongside the code to properly document the functionality and logic.
- Lines of Code (LOC) and Number of Classes (NC):**
 A positive relationship is usually observed between the number of lines of code and the number of classes (NC). As the codebase grows in size, it generally results in an increased number of classes, which reflects the structural complexity of the code.
- Number of Functions (NF) and Duplicate Code Ratio(DCR):**
 The number of functions (NF) can be associated with the duplicate code ratio (DCR). In some cases, an increased number of functions may lead to higher code duplication, especially if several functions perform similar tasks, leading to redundant code.
- Dead Code Detection and Number of Classes (NC):**
 Dead code detection is often related to metrics such as the number of functions and the number of classes. Unused or redundant functions and classes contribute to dead code, which can increase the overall size of the codebase without adding any functional value.

• Documentation Ratio (DR) and Code Complexity (CC):

The documentation ratio (DR) may indirectly help reduce code complexity (CC). Well-documented code enhances its readability and maintainability, thereby making complex code easier to understand and modify.

By understanding these interdependencies, developers can better balance the metrics to optimize code quality, maintainability, and efficiency. Through the case study, it becomes clear that the CodeLens tool is an invaluable resource for developers looking to improve efficiency and cleanliness of their codebases. By identifying areas for refactoring and optimization, developers can make more informed decisions about how to streamline their code, reduce technical debt, and maintain a high standard of quality throughout the software development lifecycle. The figure (6) illustrates visually the estimated impact of each extracted information from code on decision-making in software development life cycle.

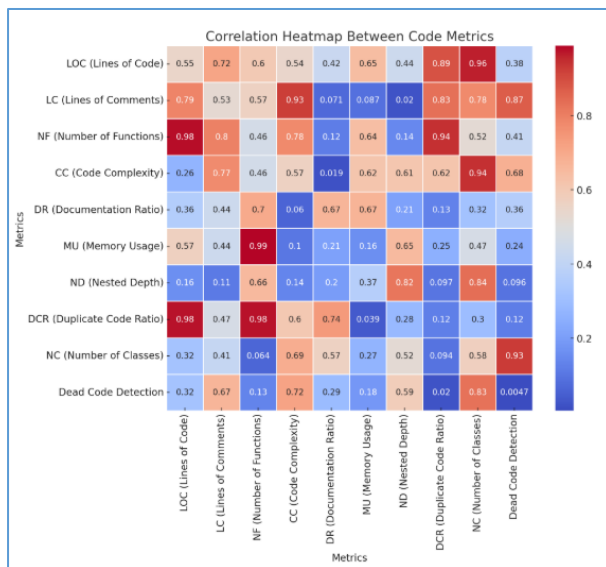


Fig. 5. The correlation between Extracted Information from code

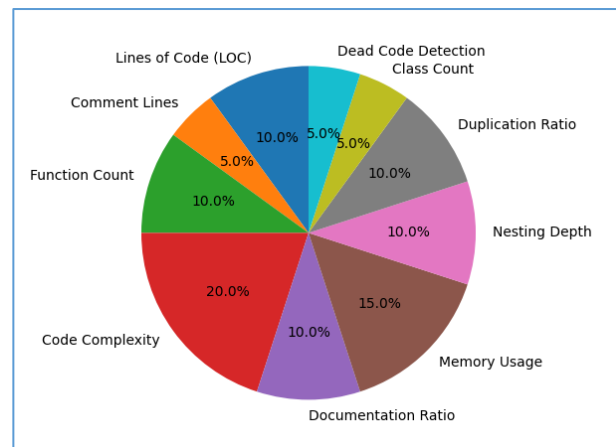


Fig. 6. Impact of Extracted Information on Decision-Making

7. Competitive Comparison of the Proposed Tool

There are many tools that help developers evaluate and analyze software quality. Table 2 provides a detailed comparison between the proposed tool and the top tools currently available in the market for this purpose. The proposed tool (CodeLens) distinguishes itself by employing development metrics which help programmers comprehend their code and make improved choices. Different metrics used in various software development phases significantly influence code size documentation modularity complexity performance reuse structure and optimization. This tool excels by integrating various software metrics which help developers achieve thorough understanding of their code. Software developers benefit from this tool because it consolidates various insights which streamline their decision-making tasks while also supporting software evaluation and enhancement.

8. Conclusion

Modern software engineering methods revolve around extracting information from software code. Using automated tools, advanced analysis tools, and visualizations helps developers make better decisions which produces maintainable, robust and high-quality software. Using such techniques not just simplifies the development process, yet helps teams to anticipate and handle challenges early on. Moreover, including such tools into the software life promotes a culture of ongoing development in which data-driven insights and real-time feedback drive innovation. Setting the basis for long-term success in a competitive market, teams could successfully minimize risks early, prioritize efforts, and guarantee that the final product satisfies business and technical objectives through concentrating on metrics that matter. With the intention of obtaining a better idea what's happening, evaluating any features

regarding a software product, estimating the quality of software and lastly track the project for ensuring that everything is under control, the suggested tool CodeLens collecting data on some features of software products. Future studies can investigate enhancing the tool's capacity to support other programming languages and frameworks, hence facilitating more general adoption over several development environments. Furthermore improving its value would be including ML methods for predicting possible code vulnerabilities and optimization opportunities.

Table II. Comparison between the proposed tool and the top tools available in the market

No.	Tool	Features	Strengths	Weaknesses
1.	SonarQube	Static code, analysis, bug detection, security vulnerabilities, and code smells.	Supports multi-language, easy to use, integrates with CI/CD tools.	Requires initial setup, slow for large projects.
2.	ESLint	Analyzes JavaScript/ TypeScript code, identifies errors, and auto-fixes formatting issues.	Lightweight and fast, highly customizable, supports auto-fixing.	Limited to JavaScript and its variants, requires initial configuration.
3.	Checkmarx	Security-focused code analysis, detects vulnerabilities.	Strong focus on security, supports multi-language, integrates with development tools.	Expensive, user interface can be complex.
4.	PMD	Static code analysis for Java, JavaScript, XML, and more.	Open-source, supports multi-language, easy to integrate.	Limited in detecting complex issues, requires manual configuration.
5.	Veracode	Security-focused code analysis, detects vulnerabilities.	Supports multi-language, provides detailed reports, integrates with CI/CD.	Expensive, can be slow in analysis.
6.	CodeClimate	Code quality analysis, measures complexity, and manages technical debt.	User-friendly interface, integrates with GitHub, supports multi-language.	costly for large projects, limited security analysis.
7.	Coverity	Static code analysis, detects bugs and security vulnerabilities.	Accurate in detecting issues, supports multi-language, integrates with development tools.	Expensive, requires complex setup.
8.	Bandit	Security-focused code analysis for Python.	Open-source, lightweight and fast, focuses on Python security.	Limited to Python, does not support other languages.
9.	Codacy	Code quality analysis, detects errors, and manages technical debt.	User-friendly interface, integrates with GitHub/GitLab, supports multi-language	costly for large projects, limited security analysis.
10.	CodeLens (my tool)	Static code analysis based on 10 key metrics.	Free and open-source, user friendly interface, Supports Java and Python, Provides comprehensive code quality analysis, detecting common issues like dead code and high complexity, Easy to integrate with development tools.	limited security analysis.

Acknowledgment

The author is very grateful to University of Mosul/ College of Computer Sciences and Mathematics for their provided facilities, which had been helpful in improving the quality of this work.

References

- [1] M. Kuutla, M. Mäntylä, U. Farooq, and M. Claes, "Time pressure in software engineering: A systematic review," *Inf. Softw. Technol.*, vol. 121, p. 106257, 2020.
- [2] F. Provost and T. Fawcett, "Data science and its relationship to big data and data-driven decision making," *Big Data*, vol. 1, no. 1, pp. 51-59, 2013.
- [3] S. Chowdhury, R. Holmes, A. Zaidman, and R. Kazman, "Revisiting the debate: Are code metrics useful for measuring maintenance effort?," *Empir. Softw. Eng.*, vol. 27, no. 6, p. 158, 2022.
- [4] T. Maaitah, "The role of business intelligence tools in the decision making process and performance," *J. Intell. Stud. Bus.*, vol. 13, no. 1, pp. 43-52, 2023.
- [5] F. Moriconi, "Improving software development life cycle using data-driven approaches," PhD diss., Sorbonne Univ., 2024.
- [6] D. T. G. Neto, T. V. Brugni, F. C. Galdi, and J. C. R. Prates, "EVA and EBITDA: How such metrics can help in the investment decision-making process," *Adv. Sci. Appl. Account.*, pp. 009-021, 2024.

-
- [7] M. A. A. Mamun, C. Berger, and J. Hansson, "Effects of measurements on correlations of software code metrics," *Empirical Software Engineering*, vol. 24, pp. 2764–2818, 2019. doi: 10.1007/s10664-019-09714-9.
 - [8] M. Zagane, M. K. Abdi, and M. Alenezi, "Deep Learning for Software Vulnerabilities Detection Using Code Metrics," *IEEE Access*, vol. 8, pp. 74562-74570, 2020. doi: 10.1109/ACCESS.2020.2988557.
 - [9] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1161-1173, May 2021. doi: 10.1109/ICSE.2021.
 - [10] A. Duque-Torres, D. Pfahl, C. Klammer, and S. Fischer, "Using source code metrics for predicting metamorphic relations at method level," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1147-1154, Mar. 2022.
 - [11] C. Park, S. Y. Moon, and R. Y. C. Kim, "Quality Visualization of Quality Metric Indicators based on Table Normalization of Static Code Building Information," *KIPS Transactions on Software and Data Engineering*, vol. 12, no. 5, pp. 199–206, May 2023.
 - [12] 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, p. 112179, 2024. doi: 10.1016/j.jss.2024.112179.
 - [13] Y. Huang, J. Huang, X. Chen, and Z. Zheng, "Towards improving the performance of comment generation models by using bytecode information," *IEEE Transactions on Software Engineering*, 2024. doi: 10.1109/TSE.2024.3523713.
 - [14] Z. Jiang, D. Wang, and D. Rao, "Leveraging meta-data of code for adapting prompt tuning for code summarization," *Applied Intelligence*, vol. 55, p. 211, 2025. doi: 10.1007/s10489-024-06197-0.
 - [15] M. A. Akbar, K. Smolander, S. Mahmood, and A. Alsanad, "Toward successful DevSecOps in software development organizations: A decision-making framework," *Inf. Softw. Technol.*, vol. 147, p. 106894, 2022.
 - [16] K. Li, A. Zhu, P. Zhao, J. Song, and J. Liu, "Utilizing deep learning to optimize software development processes," *arXiv preprint arXiv:2404.13630*, 2024.
 - [17] S. M. D. A. C. Jayatilake and G. U. Ganegoda, "Involvement of machine learning tools in healthcare decision making," *J. Healthc. Eng.*, vol. 2021, no. 1, p. 6679512, 2021.
 - [18] B. S. Mostafa and F. A. Alsalman, "Application project task scheduling using dolphin swarm technology," *Indones. J. Electr. Eng. Comput. Sci.*, vol. 23, no. 1, pp. 549-557, 2021.
 - [19] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana, "A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools," *J. Syst. Softw.*, vol. 171, p. 110827, 2021.
 - [20] J. Lin et al., "How can recommender systems benefit from large language models: A survey," *ACM Trans. Inf. Syst.*, 2023.
 - [21] A. A. Abdulmajeed, M. A. Al-Jawaherry, and T. M. Tawfeeq, "Predict the required cost to develop software engineering projects by using machine learning," *J. Phys.: Conf. Ser.*, vol. 1897, no. 1, p. 012029, May 2021.
 - [22] T. Karanikiotis and A. L. Symeonidis, "Towards understanding the impact of code modifications on software quality metrics," *arXiv preprint arXiv:2404.03953*, 2024.
 - [23] F. Moriconi, "Improving software development life cycle using data-driven approaches," *Ph.D. dissertation, Sorbonne Univ.*, 2024.
 - [24] F. Lejarza, S. Venkatesan, and M. Baldea, "Rolling horizon product quality estimation and online optimisation for supply chain management of perishable inventory," *Int. J. Prod. Res.*, pp. 1-24, 2024.
 - [25] K. Sahu, F. A. Alzahrani, R. K. Srivastava, and R. Kumar, "Evaluating the impact of prediction techniques: Software reliability perspective," *Comput. Mater. Continua*, vol. 67, no. 2, 2021.
 - [26] A. Asmaa'H and I. A. Saleh, "Develop approach to predicate software reliability growth model parameters based on machine learning," *Iraq J. Comput. Inform.*, vol. 50, no. 2, pp. 110-121, 2024.
 - [27] V. Cortellessa and D. Di Pompeo, "Analyzing the sensitivity of multi-objective software architecture refactoring to configuration characteristics," *Inf. Softw. Technol.*, vol. 135, p. 106568, 2021.
 - [28] Z. K. Al-Isawi and N. A. Al-Saati, "Selecting the best control strategies for risk management using swarm intelligence," in *Proc. Al-Sadiq Int. Conf. Commun. Inf. Technol. (AICCIT)*, 2023, pp. 310-313.
 - [29] A. H. Ali and N. N. Saleem, "Design and implementation of a testing tool (DFT-Tool) based on data flow specification," *Comput. Integr. Manuf. Syst.*, vol. 29, no. 7, pp. 1-14, 2023.
 - [30] K. S. Sieber and J. G. García-Donas, "Population affinity estimation on a Spanish sample: Testing the validity and accuracy of cranium and mandible online software methods," *Legal Med.*, vol. 60, p. 102180, 2023.
-