UKR Journal of Multidisciplinary Studies (UKRJMS)
Homepage: https://ukrpublisher.com/ukrjms/
Email: submit.ukrpublisher@gmail.com
Volume 2, Issue 11, 2026
ISSN: 3049-432X (Online)

Development of an Enhanced Automated Software Complexity Measurement System

Rilwan Adedoyin AJIYOBIOJO[1]; Stephen Olatunde OLABIYISI[2] & Christopher Akinwale OYELEYE[3]

[1,2]Department of Computer Science, Ladoke Akintola University of Technology, Ogbomoso, Oyo State, Nigeria.
[3]Department of Information Science, Ladoke Akintola University of Technology, Ogbomoso, Oyo State, Nigeria.

***Corresponding Author**: Rilwan Adedoyin AJIYOBIOJO
DOI: *https://doi.org/10.5281/zenodo.24186044*

Article History
Original Research Article
Received: 12 September 2026
Accepted: 20 October 2026
Published: 13 November 2026
Copyright © 2026 The Author(s): This is an open-access article distributed under the terms of the Creative Commons Attribution-NonCommercial **4.0 International License (CC BY-NC 4.0),** which permits non-commercial use, sharing, and reproduction in any medium, provided the original author(s) and source are appropriately credited.
Citation: Rilwan Adedoyin AJIYOBIOJO; Stephen Olatunde OLABIYISI & Christopher Akinwale OYELEYE. (2026). Development of an enhanced automated software complexity measurement system. UKR Journal of Multidisciplinary Studies (UKRJMS), 2(11), 08-17.

Abstract

Code complexity metrics are widely used in software engineering to assess quality attributes such as testability, reliability, maintainability, and performance through automated analysis of source code. Existing tools, such as the Quality Assurance of C (QA-C) system, are primarily designed for C source code on the Solaris platform and do not provide mechanisms for measuring defect rates, thereby limiting comprehensive quality assessment. This work therefore developed an enhanced automated software complexity measurement system capable of evaluating complexity and computing defect rates for C-family programming languages.The study implemented Source Lines of Code (SLOC), McCabe Cyclomatic Complexity, and Halstead Complexity metrics. The system was developed using the Waterfall Model, comprising requirements analysis, design, implementation, testing, and maintenance. It was implemented with Visual Studio 2022 IDE, C#, .NET Framework, Microsoft Roslyn Abstract Syntax Tree (AST), and SQLite database. The Roslyn AST parsed source code structures, while the SQLite History Database stored analysis records, timestamps, and computed metrics.The system was evaluated using Black-box testing on C++, C, Python, C#, and JavaScript. It achieved functionality and quality scores of 100% for C++, C, and JavaScript, and 75% for Python and C#, with an overall average of 90%. It also successfully computed defect rates. The findings show the system is effective, reliable, and user-friendly, and can serve developers, testers, and quality assurance teams in improving software quality.

Keywords: *Software Complexity, SLOC, McCabe Cyclomatic Complexity, Halstead Complexity, Automated Measurement, C#, Black-Box Testing.*

1. Introduction

Computational Complexity hypothesis focuses on identifying computational issues as claimed by their built-in problems and correlate these categories to one another (Sanjeev and Barak, 2009). These issues are task resolved by a system using other problem-solving operations mainly by a computer (Sanjeev and Barak, 2009). One of the functions of these hypothesis is establishing feasible limitation of what the computer can execute (Sanusi et al 2020).

In software lifetime development, one of the significant concerns is software complexity. Source code complexity metrics are used to measure the quality of software properties. Software complexity in order words predict analytical information about reliability, testability, and maintainability of program source code from the automatic measurement of the code (Olabiyisi *et al.* 2013). A complex code will therefore be difficult to develop and maintain by the software developer. Recently, it is important for software engineering to correctly predict the complexity of a program source code to save millions of people in maintaining time and effort (Silva *et al.* 2012).

A software metric can be defined as the quality of measuring the level to which a software system or operation possesses some attribute (such as their defect rates). However, metrics and measurements are often used interchangeably, although they have distinct meanings. Metrics can be regarded as the basis or defined measures used to quantify specific attributes, while measurements

represent the values obtained through the application of these metrics (Sanusi et al., 2020). According to the literature, quantitative measurement plays an important role across scientific disciplines, which has led to continued efforts by computer science researchers and practitioners to apply similar quantitative approaches to software development and evaluation (Lincke et al., 2008).

The software complexity metrics area of research has grown up out of two needs (Fenton and Bieman, 2014). One is the requirement for software developers to be able to control the software development process. For example, developers need to control the resources or time needed to deliver a software product or whether the software product is good enough to be released. The other need comes from researchers, who want to objectively define and measure software attributes so as to get a good understanding of software engineering (Fenton and Bieman, 2014). This software metrics field is constantly changing. Therefore, as new measures are proposed, there is no standard set of metrics. Software metrics analyst have to adjust their existing parser tools in order to accommodate the new measures. This is as being a challenge to developers since such tools usually have a very complex source code (Scotto *et al.* 2004).

Some software development practitioners point out that simplistic measurement can be harmful than its quality (Kaner, 2004). Also, other practitioners noted that metrics have suit an essential part of the software development process (Binstock, 2010). These effect of measurement on programmer's psychology has lifted concerns for harmful effects to performance anxiety, performance due to stress, and attempts to avoid the metrics, while others concluded that it has a useful effect on developers' value towards their own tasks, and stop them from being undervalued. Some disagree that the definition of many measurement methodologies is imprecise and as a result it is often unclear how some tools for computing them arrive at a particular result (Lincke *et al.* 2008), while others disagree that defective quantification is more desirable than none (You cannot control, what you cannot measure.). Evidence showed that code complexity metrics are being widely used by government agencies, National Aeronautics and Space Administration (NASA), Information Technology (IT) consultants, academic institutions (Lincke *et al.* 2008).

Software development has become an essential aspect of modern computing, requiring developers to produce high-quality and reliable software systems. Despite this importance, many software developers pay limited attention to evaluating the functionality and quality of their source code using software metrics. As a result, defects and fault-prone sections of code may remain undetected during the development process. The inability to identify such problematic code early often leads to increased maintenance costs and reduced software quality. Software metrics provide valuable insights into code characteristics such as complexity, maintainability, reliability, and efficiency. However, many developers lack adequate tools and techniques to effectively measure and interpret these metrics. According to Sanusi *et al.* (2020), controlling software development projects through measurement has become an area of growing interest in software engineering. The growing adoption of fixed-price software contracts and penalty clauses has further increased the need for accurate assessment of software quality and development performance. In addition, modern software development environments demand greater accountability and effective project control through measurable standards. Lincke *et al.* (2008) observed that the software engineering field still requires improved tools, techniques, methods, education, and training to enable developers to effectively evaluate code complexity and software quality attributes. Therefore, this research seeks to address these challenges by developing an automated approach for measuring software complexity and identifying fault-prone code that requires refactoring.

2. Related Works

Zhang and Baddoo (2007) studied the performance of three complexity metrics. They selected McCabes Cyclomatic Complexity, Halstead's Complexity, and Douces Spatial Complexity. Their experiment is established on four hypotheses using dataset from Eclipse JDT which is an open-source application. Sequel to the result, they conclude that the three-complexity metrics show different performance results during their hypotheses testing, and finally they recommended combining Cyclomatic complexity metric and Halstead metrics for better decisions on software complexity.

Another study by Panovski (2008) established a new assessment of software product quality, which focused on assessing the quality of the external features of the software product, which means evaluating the behavior of the software product when implemented. In addition, the study focused on the development of the quality model (ISO / IEC 9126) at the level of software metrics. The study is based on seven samples of the software product and assess them using ISO / IEC 9126-2 quality model. In the research, Panovski (2008) concluded that external product quality attributes are an area or category that can be used, and that the metrics provided by ISO / IEC 9126-2 can be regarded as a starting point for the definition of standards, but are not ready to use in their present form. The software metrics of the software product need to be more adapted to give better information (Panovski, 2008).

Borchert (2008) examined the technique of code profiling by using a static analysis. The study was done on 19 industrial samples and 37 samples of students' programs. He has analyzed software samples through software metrics. The results of this study recommended that the code pattern could be a useful method for rapid program comparisons and quality observation in the field of industrial application and education.

Jay *et al.* (2009) compared two metrics to calculate the complexity of code. They used McCabe Cyclomatic Complexity metric and Line of code metric (LOC) to prove the stable linear relationship between these two techniques. They used five NASA datasets with different programming languages, such as C, C++, and Java, to be the dataset for their research.

Bhatti (2010) explored the research area occupied by the software complexity metrics. He made use of a publicly available QA-C tool to automatically measure software metrics on the code written in C programming language by expressing the association between software metrics and the complexity of the source code. He also attempted to evaluate the values of these metrics graphically only, without considering the quality features and threshold limits relationship (Bhatti, 2010).

According to Silva *et al.* (2012) who worked on the relevance of three software complexity metrics: McCabe's Cyclomatic complexity, Halstead's complexity, and Shao and Wang's cognitive functional size. In the research work, 10 different programs of the same Java programming language were used and evaluate which one of the three complexity metrics is most suitable for software manufacturing. In order to calculate manually the ranking of the ten programs complexity based on the-three metrics, quota sampling method was applied by selecting five different companies and randomly asked six programmers from each company to rank the complexity of the ten programs.

Jian *et al.* (2017) studied the use of Convolutional Neural Network (CNN) to predict defects in software. This work focused only on file level prediction. Jian *et al.* (2017) claim that programs have well-defined syntax and rich semantics hidden in the Abstract Syntax Trees, which traditional features often fail to capture and the traditional methods does not really give satisfactory results.

Yu *et al.* (2018) then developed a system with a combination of derived metric sets to improve prediction of defect in concurrent software programs using deep learning technique which was named ConPredictor.

Hence, this research studied and investigated code-based complexity metrics, Source Lines-of-Code metrics (SLOC), Halstead, and McCabe Cyclomatic Complexity metrics, and designed a system that automatically measures the software complexity of C-family programming languages as well as compute the defect rate.

3. Materials and Methods

3.1 Research Approach

The Waterfall Model of software engineering was adopted in this research. It is a linear and sequential software development methodology in which each phase must be completed before proceeding to the next phase. This approach ensured a systematic and organized development process for the Automated Software Complexity Measurement System. The phases involved are as follows:

i. **Requirement Engineering:** This phase involved gathering and analyzing the requirements needed for the development of the proposed system. Relevant information was collected from websites, e-books, journals, and existing software metrics applications. The study focuses on the use of Source Lines of Code (SLOC), McCabe Cyclomatic Complexity Metrics, and Halstead Complexity Metrics to obtain objective and reproducible measurements that support software quality assurance and debugging management.

ii. **System Design:** The requirements gathered during the previous phase served as the basis for designing the system architecture, database structure, and user interfaces. The design focused on developing a system capable of measuring software complexity, computing defect-related metrics, storing analysis records, and providing facilities for retrieving historical analysis data.

iii. **System Coding (Implementation and Development):** This phase involves the actual development of the system using C# programming language and the .NET Framework. SQLite was used as the database management system. In addition, Microsoft Roslyn was integrated as the Abstract Syntax Tree (AST) analysis engine to provide accurate source-code analysis and metric computation. The system also includes a History Module for storing and managing previously analyzed records.

iv. **System Testing:** The developed system underwent rigorous testing to ensure that all components functioned correctly. Validation mechanisms were implemented to prevent invalid data entry and ensure system reliability. All interfaces and modules were tested thoroughly, including exception handling procedures. Black-box testing was adopted in this research because it focused on

evaluating the functionality of the system without considering its internal implementation details.

v. **System Implementation:** At this stage, the completed system was deployed for use. The system utilizes a SQLite database through Microsoft.Data.Sqlite to store software analysis results. Each analysis record contains the timestamp, filename, source code, and computed complexity metrics. Automatic schema migration is implemented to ensure that the database structure remains updated without requiring manual modifications. The History Module enables users to search, filter, delete, and reload previous analysis records, thereby facilitating effective software complexity analysis and comparison.

vi. **Maintenance:** Maintenance activities were carried out throughout the system lifecycle to ensure continuous improvement and reliability. Corrections, updates, and enhancements were made whenever new requirements, software metrics, or technological improvements emerge. The source code and database components were maintained to ensure optimal system performance and accuracy.

3.2 Requirement Engineering

This involves gathering necessary information in software engineering field taking software complexity metrics as a general topic to research on and having an insight of existing software metrics applications. The requirements gathered were then used in the system design stage to make the developed system user-friendly as good as possible. The system was developed with the goal of automated measurement of software complexity and increasing software reliability by measuring its defect rate. A defect rate is the percentage of output that fails to meet a quality target and also calculated by testing output for non-compliance to a quality target (John, 2017). The following formula can be used to calculate defect rate;

$$Defect\ rate = \frac{Number\ of\ Delivered\ bugs}{Lines\ of\ code} \times 100 \qquad (1)$$

Code defects are commonly measured as defects per one thousand lines of code, and can be calculated with the following formula;

$$Defects\ per\ thousand\ lines\ of\ code = \frac{Number\ of\ Delivered\ bugs}{1} \times \frac{\text{lines of code}}{1000} \qquad (2)$$

The requirements needed to gather for every analysis that were carried out from the software being developed take a main requirement which was a source project or source code for another C# or other C-family programming languages where the software application run some metrics and calculations with respect to the source code supplied. Figure 1 represents the block diagram for the developed system to work effectively and carry out a particular task.

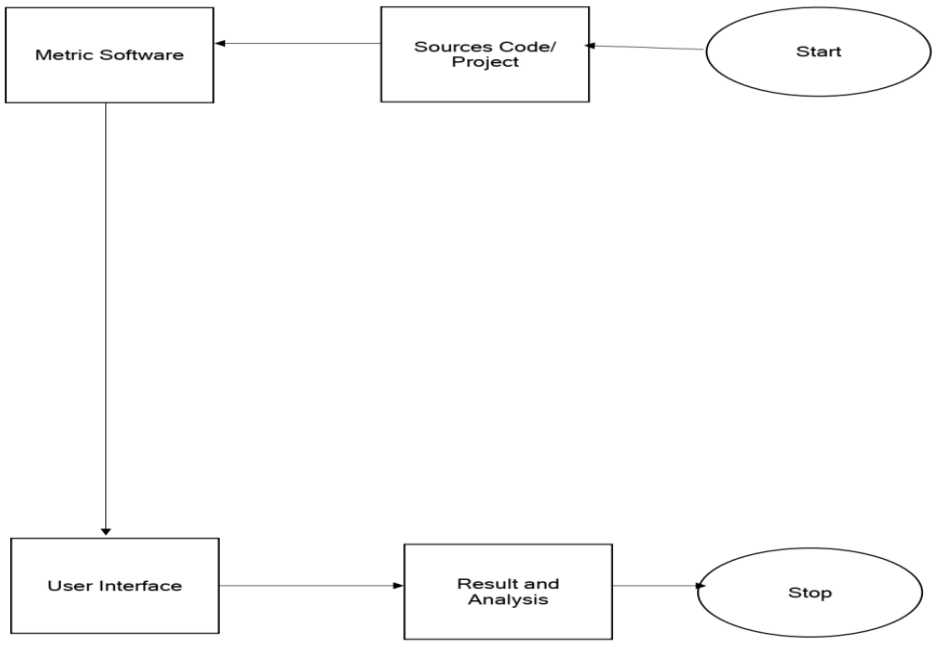

Figure 1: The Block Diagram of Software Metric Application

3.3 System Design

The software metrics system application was designed using the gathered specifications and requirements. The system followed a flow that every user could follow. This was simply used in the software engineering field when calculating the quality and functionality of software according to software metrics. The system flowed from the homepage or landing page and responded to an action when called by the user. Figure 2 represented the user flow for different user types when the system was used.

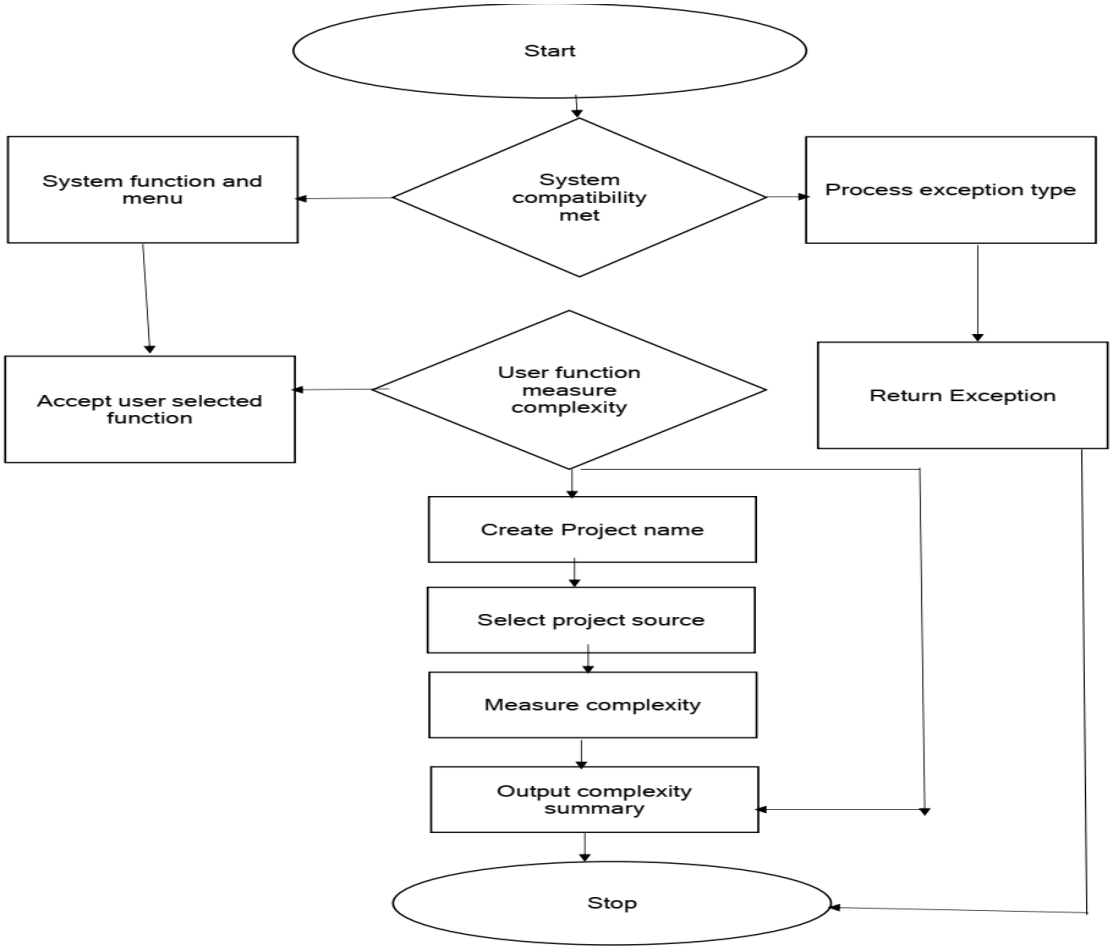

Figure 2: The Flowchart of the Developed System

3.4 System Coding

The actual development of the system was carried out using the C# programming language and the .NET Framework. SQLite was used as the database management system. In addition, Microsoft Roslyn was integrated as the Abstract Syntax Tree (AST) analysis engine to provide accurate source-code analysis and metric computation. The system also included a History Module for storing and managing previously analyzed records. C# is a multi-paradigm high-level programming language which supports several programming paradigms. The system was developed on a 2.11 GHz Intel Core i7 system running Windows 11. The final product was deployed as an executable package that contained all files required for easy use of the developed software application on any system.

The software could run on any Microsoft Windows Operating System (OS), with a minimum of Windows Vista and a maximum of Windows 11, where the executable file was generated and installed on the desired computer system intended to run the software application only, as it was another limitation that a version had not been developed for Mac and other operating systems.

i. System: PC with Intel Atom or greater, or another comparable processor, 1.6 GHz or greater configuration

ii. RAM: 1GB or greater memory

iii. Operating System: Windows Vista, Windows 7, Windows 8, Windows 10, Windows 11.

iv. Free Hard Drive Space: Up to 10 Mb for installation 20 Mb+ for use by the developed software application (this additional space is meant for storage of user-generated components).

3.5 Performance Evaluation

In order to improve the functionality and quality of the developed software, Black-box testing was adopted, which was simply a method of software testing that evaluated the functionality of an application without looking into its internal structures. This testing method could be applied to virtually every stage of software testing: unit testing, integration testing, system testing, and acceptance testing. However, the test was conducted with respect to the requirements and functionality; that is, when an input was supplied, if the output matched the desired results, then the software package was considered to have been tested successfully. Figure 3 showed the representation of black-box testing.

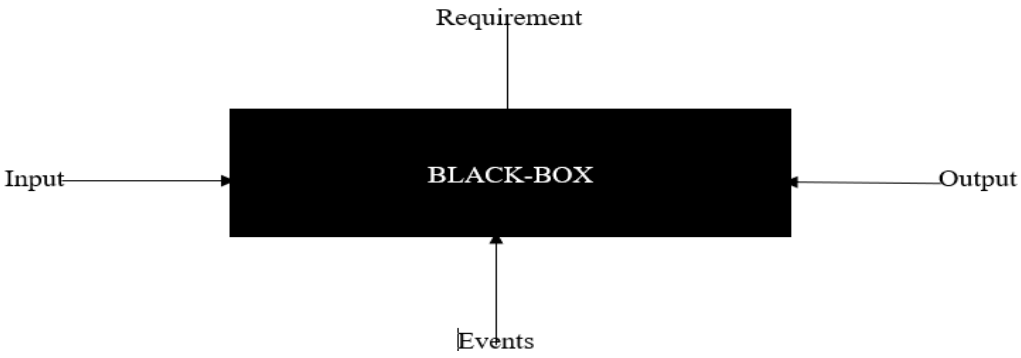

Figure 3: The Black-Box Testing

Testing was automated where it was feasible. Test plans and harnesses were carefully designed to test all methods. For each method, complete branch testing was performed where applicable. Random testing was performed where feasible and where time permitted. Test plans, containing inputs and desired outputs, were created for each unit test. All test plans were saved in files to facilitate regression testing. The user interface of the developed software was tested to ensure that each interface component performed properly.

However, in order to facilitate testing of the interface, a checklist was prepared, which contained all the functional interface components and their expected actions. Finally, the software application was system-tested by using it to create sample software components and projects. The software testers were given no specific instructions on how they should test the application, but they were encouraged to look for ways to break the application so that unanticipated bugs could be discovered and corrected. In this research, the net satisfaction index, ranging from 0 to 100%, was discussed and agreed upon with the software developers during the testing process. The net satisfaction index rating scale mapped ratings between completely dissatisfied and completely satisfied to numbers between 0 and 100%, as shown in Table 1.

Table 1: The Net Satisfaction Index Ranges.

Net Satisfaction Index Rating	**Remark**
100%	Completely Satisfied
75%	Satisfied
50%	Neutral
25%	Dissatisfied
0%	Completely Dissatisfied

4. Results and Discussion

The selection of the measured source code was made after consulting with some software developers. Each software developers have different view about their software package before the complexity measurement. As stated, code-based metric Source Lines of Code metrics (SLOC), Halstead Software Science metrics and McCabe Cyclomatic Complexity metrics are used during the development of this tool. However, the implementation of each complexity metric is discussed in the below sub-sections.

4.1 Source line of code metric (SLOC)

In evaluating the Source Line of Code metric (SLOC), this tool automatically counts and calculate the number of Blank Lines of Code (BLOC), Comment Lines of Code (CLOC) and Source Lines of Code (SLOC) which includes the actual code (logic and computation) and declarations in the source code while BLOC and CLOC improves readability and helps in understanding the code during maintenance. SLOC metric is used to calculate the quantitative attributes of the program source code and gives the software size estimations. However, in increasing the software package reliability the defect rate was measured with the calculated number of delivered bugs and the lines of code.

4.2 Halstead metrics

In evaluating the Halstead set of metrics, this tool counts the distinct number of operators, a distinct number of operands, the total number of operands, and the total number of operators in a program source code. Considering a C program, examples of unique operators are main, int, scanf, &, =, +, /, printf and so on. Also, examples of operands are a, b, c, 3, avg, %d among others. The tool searches for all the distinct operands and operators in the software package and counts how many times each distinct operator or operands appears in the source code, then calculate the total and gives the output as the total number

of operands and operators. Once these results have been computed then the values are used to calculate the Halstead Complexity metrics as stated in each formula discussed in the previous chapter. The accuracy of the tool is determined while evaluating every possible place in the software package that could have a predefined operator.

However, SLOC metric is typically applied to only one particular aspect of the source code. In contrast, the Halstead set of metrics applies to several aspect of a program source code as it is good to have more than one metric for the quantitative measure of a program and not to fully rely on one. Although, Halstead set metrics are difficult to compute manually as it is not easy to count all the operands and operators if a program is using a large number of operands and operators. Hence, in this research automatic measurement of source code is achieved by developing a tool that solves the limitations of the existing tool. Table 2 represents the summary of the measured source code.

4.3 McCabe cyclomatic complexity

McCabe is a measure of the complexity of a module's decision structure. In evaluating the McCabe Cyclomatic Complexity metrics, this developed tool looks for IF, FOR, WHILE, CASE statements in the source code and counts them to calculate the McCabe complexity value. It counts the number of decision or control structures and maps a program to a directed, connected graph. The branches of the graph are regarded as decision or control statements and the edges indicate a control path that define the program flow. Nevertheless, due to the fact that no organization or software company will want to release their source code and because of the license time restrictions two senior software programmers were consulted. The selected source code for performing the code analysis is Python and C#.

The software developer's opinion about Python was that the package is well structured and designed which should possibly contain less complex code. In other words, the software package C# was supposed to be more complex as said by the software developer. Hence, the McCabe Cyclomatic metric used in this tool results for a fast assessment and from the calculated results the Python software package has less complex code while the C# software package has more complex code which relates to the software developer's opinion. Figure 4 represents the calculated result of the Python software package and Figure 5 shows the calculated result of the C# software package.

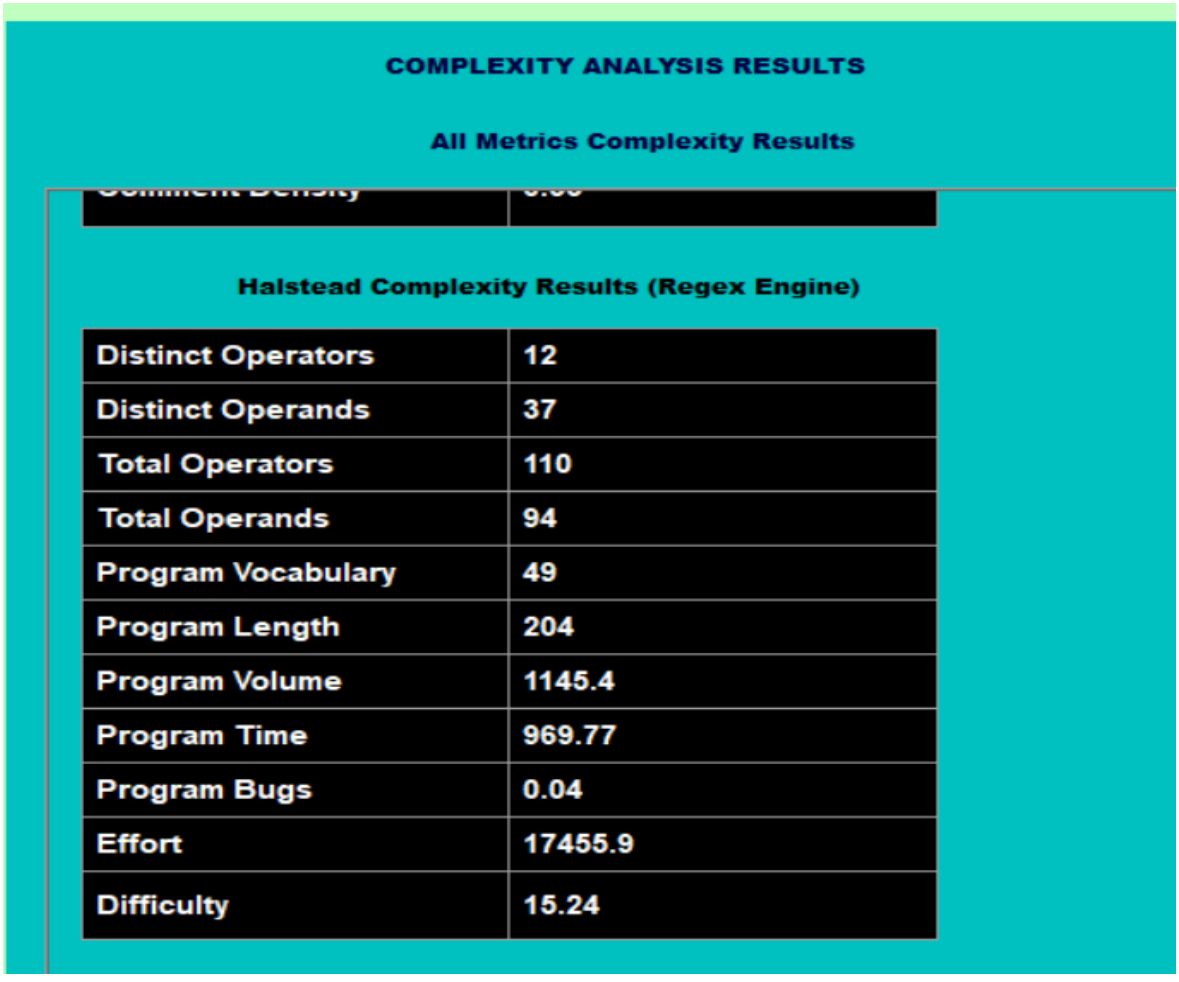

Figure 4: Desktop Interface Showing Halstead Complexity Result (Regex Engine)

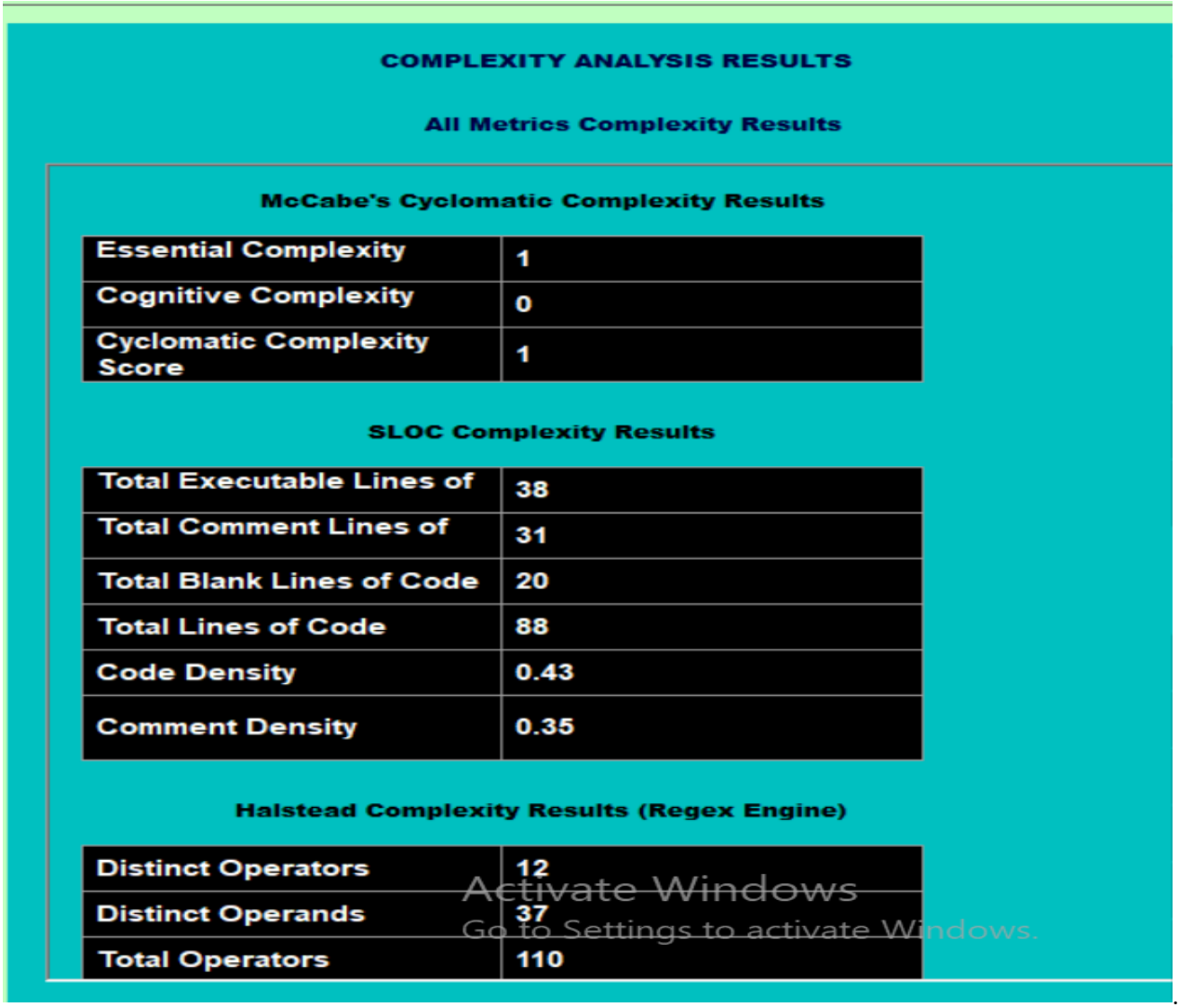

Figure 5: Desktop Interface Showing All Analysis Results

4.4 Evaluation of the Tool

The developed tool has a view in which the effectiveness of the waterfall model is being implemented. The outcome of the developed tool shows that it has the ability to perform automatic measurement of source code on a regular basis and also extend the functionality of the existing tool. This developed tool has made it possible for software testers to indicate which part of the code needs refactoring and stores the generated report in the database so as to monitor the code changes between the releases. Also, this tool has the functionality to check up to 500,000 lines of code.

However, black-box testing method was used to test the tool based on requirements and functionality by the software developers because it is a software testing method which evaluates the functionality of an application without looking into its internal structure or workings that is, in the testing method, the structure and design of the source code are not disclosed to the software tester, software developers and end-users perform this test on software.

The software programmers were chosen in the software testing phase and the test was conducted on the system. The ratings are clearly stated in Table 2 showing the software developer's opinion about the developed tool. Hence, this black-box testing method attempts to find errors in functionality and quality of the developed system as represented in Figure 7. The functionality is performed by the end-users on providing the input and the output equals the desired results while the quality is the level to which the system meets specified requirements. The overall average for functionality and quality of the developed system results to 90% and 90% respectively.

Table 2: Summary of Black-Box Testing for Each Software Developer

Programming Language	Functionality	Quality
C++	100%	100%
c	100%	100%

Python	75%	75%
C#	75%	75%
JavaScript	100%	100%
Average	90%	90%

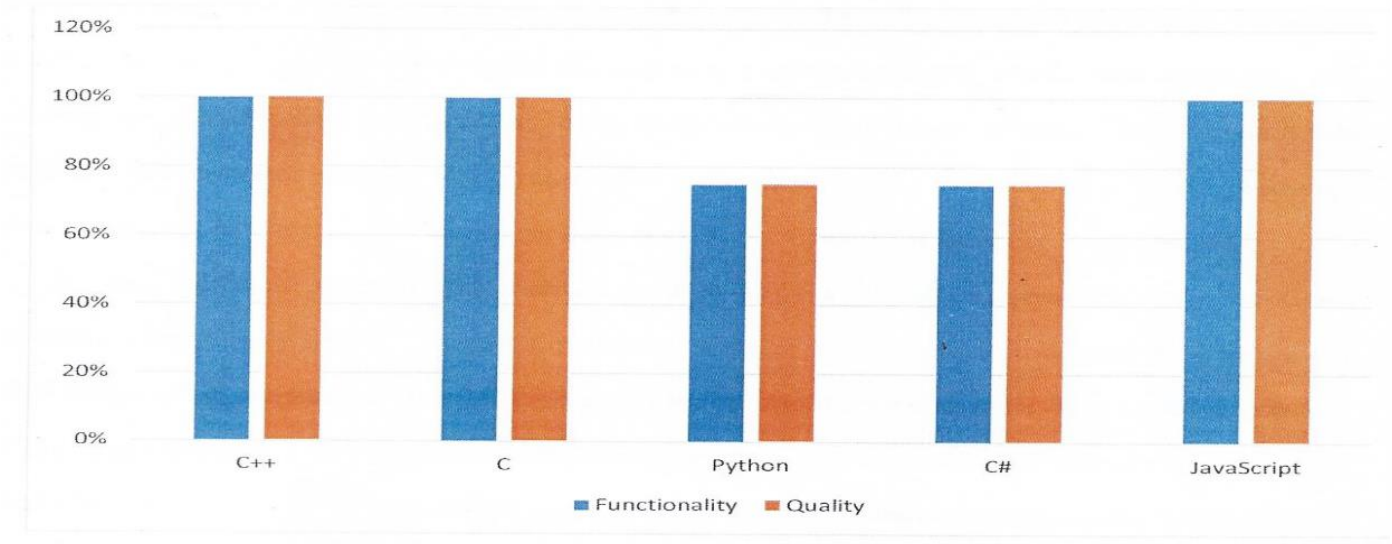

Figure 7: Summary of Black-Box Testing for Each Software Developer

5. Conclusion

In this research, code-based metrics Source Line of Code metrics (SLOC), McCabe Cyclomatic Complexity and Halstead Complexity Metrics were studied and used to achieve the goal of developing an automatic software complexity measurement tool. The metrics provide visibility and control for the complex software development procedure and therefore, they are a valuable tool for providing guidance on making the software development process better as well as meeting organizational goals to enhance software productivity and quality.

This tool was successfully developed and tested efficiently to run on Microsoft Windows operating system. However, the software testing is a significant stage of the software development life-cycle. In order words, it is necessary to optimize test cases and generate them automatically so as to reduce testing time, effort and cost. Hence, the black-box testing method was used in this research because it focuses on evaluating the functionality of the system.

In the completion of this research, the under-listed are recommended;

i. Software developers and researchers should endeavor to develop new custom metrics using some of the existing ones such as McCabe Cyclomatic Complexity metrics, Halstead Complexity Metrics and Source Lines of Code metrics.

ii. The tool should be improved on by identifying a plain text from a code once it is uploaded.

iii. The gathered metrics should be evaluated carefully to identify possible interconnection between certain metrics

iv. Software teams can use software project metrics to adapt workflow and technical activities.

References

1. Bansiya, J. (2000). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering, 28*(1), 4–17.
2. Basili, V. R., & Weiss, D. M. (2003). A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering, SE-10*, 728–738.
3. Bhatti, H. R. (2010). *Automatic measurement of source code complexity* [Master's thesis, Luleå University of Technology].
4. Binstock, A. (2010). *Integration watch: Using metrics effectively*. SD Times. http://www.sdtimes.com/link/34157
5. Borchert, T. (2008). *Code profiling: Static code analysis* [Master's thesis, Karlstad University].
6. Fenton, N., & Bieman, J. (2014). *Software metrics: A rigorous and practical approach* (3rd ed.). CRC Press.
7. Jay, G., Hale, J. E., Smith, R. K., Hale, D., Kraf, N. A., & Ward, C. (2009). Cyclomatic complexity

and lines of code: Empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications*, 7–14.

8. Jian, L., Pinkie, H., Jieming, Z., & Micheal, L. (2017). Software defect prediction via convolutional neural network. *IEEE International Conference on Software Quality, Reliability and Security*, 8–10.

9. Kaner, C. (2004). *Software engineer metrics: What do they measure and how do we know?* http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.2542

10. Lincke, R., Lundberg, J., & Lowe, W. (2008). Comparing software metrics tools. *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*, 131–142. ACM.

11. Olabiyisi, S. O., Omidiora, E. O., & Sotonwa, K. A. (2013). Comparative analysis of software complexity of searching algorithms using code based metrics. *International Journal of Scientific and Engineering Research, 4*(6), 1–2.

12. Panovski, G. (2008). *Product software quality* [Master's thesis, Technische Universiteit Eindhoven].

13. Sanjeev, A., & Barak, B. (2009). *Computational complexity: A modern approach*. Cambridge University Press.

14. Sanusi, B. A., Olabiyisi, S. O., Afolabi, A. O., & Olowoye, A. O. (2020). Development of an enhanced automated software complexity measurement system. *Journal of Advances in Computational Intelligence Theory, 1*(3), 1–11. https://doi.org/10.5281/zenodo.3597631

15. Scotto, M., Sillitti, A., Succi, G., & Vernazza, T. (2004). A relational approach to software metrics. *ACM Symposium on Applied Computing*, 1536–1540.

16. Silva, D., Kodagoda, N., & Perera, H. (2012). Applicability of three complexity metrics. *Proceedings of the International Conference on Advances in ICT for Emerging Regions*, 12–16.

17. Yu, T., Wen, W., Han, X., & Hayes, J. (2018). ConPredictor: Concurrency defect prediction in real-world applications. *IEEE International Conference on Software Testing, Verification and Validation*, 168–179.

18. Zhang, M., & Baddoo, N. (2007). Performance comparison of software complexity metrics in an open source project. *Proceedings of the 14th European Conference on Software Process Improvement*, 1–16.