Code Complexity Measuring Machine

Pinnawala A.N.
dept. Information Technology
Sri Lanka Institute of Information Technology
Kandy, Sri Lanka
it22235824@my.sliit.lk

Gunathilaka H.A.H.V..
dept. Information Technology
Sri Lanka Institute of Information Technology
Kandy, Sri Lanka
it22219916@my.sliit.lk

Shadhir M.F.M.
dept. Information Technology
Sri Lanka Institute of Information Technology
Kandy, Sri Lanka
it22237118@my.sliit.lk

Bilal R.A.M.
dept. Information Technology
Sri Lanka Institute of Information Technology
Kandy, Sri Lanka
it22548900@my.sliit.lk

***Abstract*—The "Code Complexity Measuring Machine" project aims to develop a comprehensive tool for assessing and managing software complexity. This tool integrates multiple metrics, including time complexity, cyclomatic complexity, and code size analysis, to provide a holistic view of a software system's maintainability and performance. By calculating nesting depth, cyclomatic complexity, and logical statement density, the tool offers insights into potential improvements and helps developers visualize complexity issues. Additionally, it features a collaborative environment where teams can share dashboards, engage in real-time discussions, and conduct collaborative code reviews, fostering continuous improvement. This project leverages established metrics such as McCabe's cyclomatic complexity [1] and maintainability index [2], along with modern collaborative features to enhance software development processes.**

Keywords— Code Complexity, Cyclomatic Complexity, Time Complexity, Code Size Analysis, Maintainability Index, Software Metrics, Collaborative Environment, Real-time Collaboration, Code Visualization, Software Maintainability

I. INTRODUCTION

As software systems grow in size and complexity, ensuring maintainability and optimizing performance become increasingly challenging. Code complexity is a critical factor influencing these aspects, as it directly affects the ease of understanding, testing, and modifying code. High complexity can lead to increased development time, more bugs, and higher maintenance costs. Therefore, measuring and managing code complexity is essential for maintaining software quality throughout its lifecycle.

The "Code Complexity Measuring Machine" addresses this need by offering a comprehensive set of tools to analyze various dimensions of code complexity. It focuses on key metrics such as time complexity, cyclomatic complexity, and code size, which are crucial for evaluating and improving software maintainability. Time complexity analysis provides insights into how the code's runtime scales with input size, helping developers identify and mitigate performance bottlenecks [3]. Cyclomatic complexity, a metric introduced by McCabe [1], quantifies the number of linearly independent paths through a program's source code, offering a measure of its logical complexity and testing requirements.

In addition to these traditional metrics, the project includes a detailed code size analysis, which counts logical statements and calculates the ratio of source lines of code (SLOC) to lines of code (LOC). This helps developers understand the density and structure of the code, further informing maintenance strategies [4]. The maintainability index, as described by Oman and Hagemeister [2], is also calculated, providing a single composite score that reflects the overall maintainability of the codebase.

Recognizing the importance of collaboration in software development, the project integrates features that support real-time collaboration and code review. Shared dashboards allow team members to monitor key complexity metrics collectively, while real-time collaboration tools enable simultaneous discussions and annotations, similar to platforms like Google Docs [5]. These features promote continuous feedback and improvement, ensuring that complexity is managed effectively throughout the development process.

By combining these analytical tools with collaborative features, the "Code Complexity Measuring Machine" aims to enhance the quality and maintainability of software systems, making it easier for development teams to produce robust, efficient, and maintainable code.

II. LITERATURE REVIEW

Code complexity has long been recognized as a significant factor influencing software quality, maintainability, and reliability. The concept of code complexity encompasses various dimensions, such as cyclomatic complexity, time complexity, and code size, each contributing to the overall understanding of how difficult it is to understand, test, and modify a software system.

Introduced by McCabe in 1976, cyclomatic complexity is one of the most widely used metrics for measuring the complexity of a program’s control flow. It calculates the number of linearly independent paths through a program’s source code, which correlates with the number of test cases needed to achieve full branch coverage [1]. High cyclomatic complexity indicates a more intricate and potentially error-prone codebase, as it suggests numerous decision points and complex logic paths. This metric is essential for identifying overly complex modules that may benefit from refactoring to improve maintainability.

Time complexity is a fundamental concept in algorithm analysis, often linked to the performance and efficiency of software systems. Defined as the relationship between the size of the input and the time required to execute an algorithm, time complexity is crucial for predicting how software will scale as data volumes grow. According to Aho, Hopcroft, and Ullman [3], optimizing time complexity is vital for ensuring that software remains responsive and efficient under varying loads. Tools that measure time complexity help developers understand potential bottlenecks and optimize code for better performance.

Halstead Metrics, introduced by Maurice Halstead in his seminal 1977 work *Elements of Software Science* [6], revolutionized the way code complexity was understood by providing a mathematical model based on operators and operands within a software program. Halstead's key contribution was the notion that software complexity could be quantified by analysing the basic operations and symbols used in the code, rather than relying solely on structural or algorithmic analysis. Halstead's metrics are based on simple counts of the number of unique and total operators and operands. From these values, five derived measures are calculated, including program length, volume, vocabulary, difficulty and effort. These metrics aim to predict the cognitive effort required for understanding and maintaining a software program.

Loops are one of the most computationally intensive constructs in programming, and nested loops can exponentially increase time complexity. The Nested Loop Matrix method analyses the depth and arrangement of nested loops within a program to evaluate the structural complexity. Deeply nested loops often lead to higher time complexity, increasing the computational cost and runtime of the software. The analysis of loop structures is particularly important in performance-critical applications where time complexity plays a significant role. Aho, Hopcroft, and Ullman [4] emphasize that deeply nested loops should be minimized for more efficient algorithm design. Tools that analyse loop structures help identify performance bottlenecks and suggest optimization strategies, such as refactoring loops or applying algorithmic improvements.

Lines of Code (LOC) has been a widely used metric in software engineering for measuring the size of a codebase. Despite its simplicity, LOC has proven useful in predicting development and maintenance costs. According to Kafura and Henry (1981), LOC can serve as a significant factor for estimating software project costs, particularly in large-scale systems where understanding code size can influence resource allocation throughout the project lifecycle. In their work, they argue that LOC, when combined with other metrics such as information flow, can provide deeper insights into software structure and the complexity of interactions within the codebase [7].

Kan (2002) highlights the continued reliance on LOC as an industry-standard measure of productivity, despite its well-documented limitations. While LOC may not accurately reflect the quality or complexity of code in every case, it remains a valuable indicator in many environments due to its simplicity and ease of use. Kan suggests that, when combined with other metrics like cyclomatic complexity or maintainability indices, LOC can be a practical tool for managing software quality and predicting long-term maintenance needs [8].

Halstead (1977) supports the notion that LOC is most effective when used in conjunction with other structural metrics. His research emphasizes that LOC alone may not capture the full complexity of a system but can provide valuable context when analyzing code characteristics like operational and data complexity. Halstead's formulae, which include considerations such as the number of operators and operands, underscore the need to pair LOC with other complexity measures to gain a comprehensive view of code behavior and maintainability [9].

In summary, while LOC may have limitations in isolation, the evidence from multiple sources suggests that it remains a critical metric when combined with other measures, especially for studying the structural and functional characteristics of software systems.

The size of the code, often measured in Source Lines of Code (SLOC) or Logical Lines of Code (LLOC), also plays a significant role in assessing complexity. While larger codebases are not inherently more complex, they often present challenges related to maintainability, as noted by Fenton and Pfleeger . The maintainability index, developed by Oman and Hagemeister, integrates code size with other factors like cyclomatic complexity and Halstead metrics to provide a composite score reflecting how easy the code is to maintain [2]. A lower maintainability index suggests that the code might require significant effort to update or debug, highlighting the need for ongoing code reviews and refactoring.

The Maintainability Index (MI) is a composite metric designed to provide a single, overall indicator of code maintainability. It was introduced as part of software quality evaluation frameworks and is calculated using a combination of Cyclomatic Complexity, Halstead Volume, and lines of code. Higher MI values indicate better maintainability, while lower values signal that the code may be difficult to modify or extend. According to Microsoft's documentation, MI values range from 0 to 100, with thresholds typically set to highlight critical areas requiring refactoring [10]. This metric plays a crucial role in maintainability analysis, particularly in large software projects, where long-term maintenance is a major concern.

Introduced by McCabe in 1976, Cyclomatic Complexity measures the number of independent paths through a program's source code. It quantifies the decision logic, which correlates with the difficulty of testing and maintaining the software. Cyclomatic Complexity is computed from the control flow graph of a program, where nodes represent blocks of code and edges represent control flow between them. Higher values of Cyclomatic Complexity suggest that a program is more complex and error-prone, requiring greater effort to test and maintain [11]. It is one of the most widely used metrics in modern static code analysis tools, reflecting its importance in software engineering practices.

The Halstead Metrics were proposed by Maurice Halstead in the 1970s to measure the cognitive complexity of software. Halstead Volume, one of the key metrics, is derived from the number of operators and operands in the code. This metric is particularly useful in estimating the mental effort required to understand a given piece of software. Halstead's theory suggests that larger volumes indicate higher complexity, which can negatively affect maintainability, readability, and the potential for defects [12]. The Halstead Metrics provide a more nuanced view of software complexity by examining the fundamental operations in the code rather than just structural elements.

Various tools have been developed to integrate these metrics into automated software quality checks. By using these metrics in tandem, modern complexity analysis tools provide developers with actionable insights into code maintainability, readability, and testability. These tools often highlight areas of concern and suggest improvements, helping developers manage technical

debt and ensure code quality throughout the software development lifecycle.

Collaborative code review is a vital process in modern software development, enabling teams to work together to identify issues, ensure code quality, and improve overall maintainability. In the context of the "Code Complexity Measuring Machine," collaborative code review goes beyond traditional review mechanisms by integrating complexity analysis into the review process itself. This ensures that team members can simultaneously address both functional correctness and complexity-related concerns, ultimately enhancing software robustness and performance.

The collaborative environment is built around shared dashboards that display key complexity metrics in real time. These dashboards provide a unified view of the project's status, allowing developers to monitor complexity across different dimensions, such as cyclomatic complexity, time complexity, and code size. Each developer can view the current state of the codebase and compare it with previous versions, enabling them to track the impact of their changes on code complexity. According to Weibel et al. [13], dashboards in collaborative systems can significantly improve transparency and team coordination, ensuring that all members are aligned on project goals and complexity standards.

Real-time collaboration features allow team members to simultaneously view and discuss complexity analysis results, similar to collaborative platforms like Google Docs. Multiple users can interact with the same set of code and metrics, facilitating immediate feedback and group decision-making. Real-time collaboration has been shown to enhance the speed and effectiveness of code review processes, as discussed by Storey et al. [14], by enabling multiple reviewers to contribute simultaneously and resolve issues more efficiently.

Users can add comments and annotations directly to the complexity analysis results, allowing for in-depth discussions about specific code issues. These comments are tied to particular sections of the code or metrics, ensuring that the context is preserved for future reference. This feature supports asynchronous collaboration, where developers can leave feedback for others to review at their convenience. As noted by Bacchelli and Bird [15], comments in code review are essential for providing constructive feedback and promoting understanding among team members.

The system maintains a history of discussions and decisions related to complexity issues, enabling the team to track the evolution of the codebase over time. This historical record serves as a valuable resource for understanding past decisions, the rationale behind certain design choices, and how complexity concerns were addressed. Maintaining this history has been shown to be particularly beneficial in large and distributed teams, as it helps avoid repeated discussions and misunderstandings, as highlighted by Rigby et al. [16].

Feedback loops are integrated into the collaborative environment, allowing developers to provide input on complexity findings and suggest improvements. These loops facilitate continuous feedback, where suggestions from team members are incorporated into future development cycles, promoting a culture of constant refinement. This approach aligns with the findings of Bosu et al. [17], who emphasize the importance of ongoing feedback in improving both code quality and team collaboration.

In traditional code reviews, the focus is often on functionality, style, and adherence to coding standards. However, in this project, code reviews are explicitly focused on addressing complexity metrics. This ensures that code maintainability and performance are prioritized from the outset, reducing technical debt over time. Developers can collectively analyze the complexity metrics presented on the shared dashboard and discuss strategies for reducing cyclomatic complexity, optimizing algorithms, or refactoring large code segments. This complexity-driven review process has been shown to significantly improve long-term maintainability, as documented by Kemerer [18].

In recent years, the importance of collaborative tools in managing code complexity has gained attention. Weis and Lenk [18] discuss the benefits of real-time collaborative editing, which allows multiple developers to work on the same codebase simultaneously. This approach not only facilitates immediate feedback but also enables collective decision-making regarding code structure and complexity. Collaborative code reviews, supported by shared dashboards and annotations, further enhance the ability to manage complexity by ensuring that all team members are aligned on coding standards and best practices.

Together, these metrics and collaborative tools form the foundation of the "Code Complexity Measuring Machine." By integrating cyclomatic complexity, time complexity, and code size analysis with real-time collaboration features, this project builds upon existing literature to create a comprehensive solution for managing software complexity. The goal is to provide actionable insights that help developers maintain high-quality codebases, reduce technical debt, and improve the overall efficiency of the software development process.

III. METHODOLOGY

A. System Overview

The "Code Complexity Measuring Machine" is a software system designed to assess and manage code complexity through a range of metrics and real-time collaboration features. The system is divided into several key modules that calculate metrics like Lines of Code (LOC)**,** Cyclomatic Complexity**,** and Logical Lines of Code (LLOC**)**. It provides a user-friendly interface where developers can actively collaborate on reviewing and improving code quality.

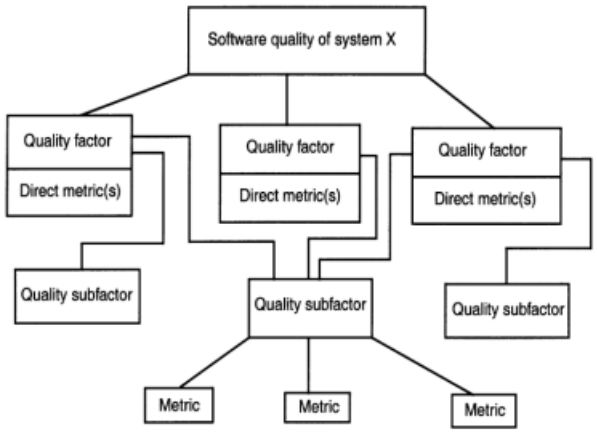

Figure 1. Software Quality Metrics

The interface features a dashboard that visualizes various code metrics through charts and graphs, allowing users to track the complexity of their code in real time. These visualizations help identify areas of high complexity that may require refactoring. Developers can also interact with a live code editor with integrated syntax highlighting, where multiple users can work on the same piece of code simultaneously. Each user is represented by a unique colored cursor, promoting seamless collaboration.

A comments section allows users to annotate the code with feedback and suggestions, supporting asynchronous reviews and team discussions. The system also provides options for downloading complexity reports and reviewed code using built-in tools.

The real-time collaboration and visualization features ensure that the code complexity analysis process is efficient, transparent, and collaborative.

B. Data Collection and Parsing

The first step involves collecting source code from the project repository and parsing it using abstract syntax trees (AST). The ASTs are used to extract relevant information for each complexity metric, including control flow constructs, algorithmic structures, and code size.

C. Integrated Complexity Metrics Calculation

After parsing, the framework calculates the cyclomatic complexity, time complexity, and code size in a single pass. Cyclomatic complexity is determined by constructing control flow graphs (CFGs), while time complexity is estimated by analyzing nested loops and recursive functions. Code size metrics, including LOC and LLOC, are computed simultaneously, and the maintainability index is derived using these metrics.

1) LOC Complexity Calculation

To calculate LOC Complexity in the "Code Complexity Measuring Machine," the methodology focuses on determining Source Lines of Code (SLOC) and Logical Lines of Code (LLOC) as key metrics for assessing code size and complexity. lang-map will be utilized to detect the programming language of the code, ensuring language-specific parsing rules are applied for accurate LOC calculations.

SLOC will be computed by counting all lines of executable code, while excluding non-functional elements such as comments and blank lines. LLOC, on the other hand, will focus on meaningful statements like control structures, function calls, and expressions that directly affect the program's behavior. The system will achieve this by parsing the source code into an abstract syntax tree (AST), allowing it to analyze the logical flow and count significant code elements.

The results will be presented through a real-time interface using React and the CodeMirror editor, which will dynamically update the LOC metrics as the user edits the code. Additionally, users can export the LOC data using file-saver and jszip, allowing for easy download and offline review. This methodology ensures that both raw code volume and logical code complexity are accurately measured and visualized.

2) Halstead Metrics Calculation

The Halstead Metrics are calculated based on the operators and operands identified during the parsing stage. Operators include arithmetic symbols, logical operators and comparison operators. Operands include variables, constants, and function names.

First, the count of operators and operands are taken from the algorithm. Distinct operators n1 is taken by counting the number of unique operators and distinct operands n2 is taken by counting the number of unique operands in the algorithm. The total operators N1 and total operands N2 are taken by counting the total number of operators and operands in the algorithm.

Once the operators and operands are counted, five key metrics are derived. Program length N, volume V, vocabulary n, difficulty D and effort E are calculated. The relevant equations to the five key metrics are given below.

$$N = N1 + N2 \quad (1)$$

Equation (1) represents the total number of elements, operators and operands in the code (N derives program length).

$$n = n1 + n2 \quad (2)$$

Equation (2) represent the total number of unique elements in the code (n derives program vocabulary).

$$V = N \cdot \log2(n) \quad (3)$$

Equation (3) represent program Volume (V) reflects the size of the codebase in terms of its information content.

$$D = n1 \cdot N2 \quad (4)$$
$$2 \quad n2$$

Equation (4) represent program Difficulty(D) measures the complexity of combining operators and operands.

$$E = D \cdot V \quad (5)$$

Equation (5) represent program Effort (E) estimates the amount of mental effort required to understand and maintain the code.

The derived metrics offer insight into various aspects of the software's cognitive complexity:

- High Volume (V): A large program volume often correlates with higher complexity, indicating a dense or verbose codebase that could require significant effort to maintain.
- High Difficulty (D): A high difficulty value points to intricate logic, with complex interactions between operators and operands. This suggests that understanding the program may require advanced knowledge or experience.
- High Effort (E): A high effort score implies that the code is cognitively demanding, requiring more time and resources to fully comprehend or modify.

3) Nested Loop Matrix Calculation

In this stage, the focus is on evaluating the structural complexity of the code by analysing loop nesting. Loops, especially nested loops, can significantly impact the performance of a program, increasing its time complexity. Nested Loop Matrix calculation is calculated by following steps:

- A Nested Loop Matrix is created, which represents the relationship between loops and their nesting depths.
- The matrix rows correspond to the nesting levels (e.g., single loop, double loop), while the columns correspond to the number of occurrences at each level of nesting.

4) Maintainability Index Calculation

The function evaluates code complexity using three well-established software metrics: Maintainability Index, Cyclomatic Complexity, and Halstead Volume. These metrics are chosen for their effectiveness in assessing various aspects of code maintainability and complexity.

a) Maintainability Index (MI):

The Maintainability Index is used to assess the ease with which code can be maintained, improved, and extended over time. It is calculated based on Cyclomatic Complexity, Halstead Volume, and the total number of lines of code. The index provides a value in the range of 0 to 100, where higher values correspond to better maintainability.

The metric originally was calculated as follows [10]:

$$MI = 171 - 5.2 \times ln(VVV) - 0.23 \times CCC - 16.2 \times ln(LOC)$$

Where:

- VVV = Halstead Volume
- CCC = Cyclomatic Complexity
- LOC = Lines of Code

The Maintainability Index is classified as follows:

- 0–9 (Red): Low maintainability (high complexity, requires immediate attention).
- 10–19 (Yellow): Moderate maintainability (code requires improvement).
- 20–100 (Green): High maintainability (code is easily maintainable).

Cyclomatic Complexity (CC): Cyclomatic Complexity measures the number of independent paths through a program. It is calculated by constructing the control flow graph of the program and determining the number of linearly independent paths. High Cyclomatic Complexity indicates a program that is harder to understand and test. The target is to minimize this value to reduce the risk of errors and enhance maintainability [11].

Halstead Volume (HV): Halstead Volume measures the cognitive complexity required to understand a program by quantifying the number of operations and operands. It reflects the information density of the code. Higher Halstead Volume indicates that more mental effort is needed to comprehend the code [12].

The system follows a client-server architecture. The user interface (UI) is developed using React, which provides an interactive environment for users to submit their code and view the complexity results. The backend, implemented in Java, processes the submitted code and computes the Maintainability Index, Cyclomatic Complexity, and Halstead Volume.

1. Input Submission: The user enters the code into a text area in the front-end, which is then sent to the backend via an HTTP POST request using Axios. The backend is hosted on a local server, and the endpoint /calculate-maintainability-index is responsible for handling the complexity calculation request.
2. Response Handling: Upon receiving the request, the backend calculates the Maintainability Index, Cyclomatic Complexity, and Halstead Volume. These values are returned as an array to the front-end, where they are displayed dynamically. The UI changes color based on the Maintainability Index values, providing an intuitive, real-time assessment of code quality:
    - Red**:** Indicates critical maintainability issues (MI 0–9).

- Yellow**:** Signals moderate maintainability issues (MI 10–19).
- Green**:** Denotes good maintainability (MI 20–100).

5) Error Handling and User Feedback

To ensure robustness, the front-end incorporates error handling mechanisms. If the complexity check fails (e.g., due to invalid input or server error), an error message is displayed, prompting the user to verify the input code. The user can also clear the input and results via the "Clear" button to reset the tool for another analysis.

6) Dynamic Feedback Using Code Visualization

The tool uses the Count Up library to present real-time, animated results for the Maintainability Index, Cyclomatic Complexity, and Lines of Code. The results are displayed dynamically on the UI, providing immediate feedback to users and improving the user experience. Code syntax is highlighted using Prism, which enhances readability and clarity.w

D. Visualization and Reporting

The results are visualized through a unified dashboard that displays complexity metrics in real-time. Developers can interact with the dashboard to drill down into specific modules or functions, view historical trends, and compare complexity metrics across different code versions.

E. Real-Time Collaboration and Feedback

The framework supports real-time collaboration by allowing multiple team members to access the dashboard simultaneously. Developers can provide feedback directly within the system, highlighting specific complexity issues or suggesting improvements. The system maintains a history of feedback and actions taken, enabling continuous monitoring and refinement of the code.

The collaborative code review feature of the "Code Complexity Measuring Machine" will be implemented using React and Socket.io to enable real-time interaction between multiple users. Socket.io will handle WebSocket connections, allowing for instant communication between the server and clients. This enables users to view changes made by others in real time, facilitating collaborative discussions and live code editing. Each user's actions, such as typing, commenting, or viewing different parts of the code, will be synchronized across all participants, ensuring seamless collaboration during code reviews.

To provide an efficient and intuitive environment for code editing, CodeMirror will be used as the core library for the in-browser code editor. CodeMirror supports syntax highlighting for multiple programming languages and integrates well with React, making it ideal for an interactive user experience. As users edit and review code, changes will be captured and shared in real time using Socket.io, enabling multiple team members to work together on the same file simultaneously. The lang-map library will be used to dynamically detect the programming language being edited, ensuring that the correct syntax highlighting, and complexity analysis rules are applied based on the language.

In addition to real-time collaboration, users will have the option to download the reviewed code or complexity reports for offline use. This will be facilitated by the file-saver and jszip libraries. Once a review session is complete, users can generate a ZIP file of the code and its corresponding complexity analysis results. The file-saver library will enable users to download files directly from the browser, while jszip will compress multiple files into a single package, ensuring efficient file management and distribution. This combination of real-time interaction and offline functionality ensures that the collaborative code review process remains flexible, allowing teams to work both synchronously and asynchronously as needed.

F. Implementation

The unified framework is built using a combination of Java for the backend server and JavaScript with React for the front-end interface. The system is integrated with version control systems like Git, allowing it to automatically update complexity metrics as the codebase evolves. Real-time collaboration features are implemented using Socket.io, enabling seamless communication and live updates between users.

1) Real-Time Collaboration with Socket.io

Real-time collaboration is a critical component of the system, implemented using Socket.io. This library allows seamless, bidirectional communication between the server and clients, ensuring that updates are reflected immediately for all users.

Whenever a user interacts with the code editor (e.g., edits code, adds comments, or performs reviews), these actions are sent via a WebSocket connection to the server. The server then broadcasts these changes to other connected users, enabling them to see the updates in real-time. This feature is particularly useful for collaborative code reviews, where multiple users can simultaneously review and comment on code complexity.

2) File Handling with File-Saver

In this system, the File-Saver library is used to allow users to upload, handle, and interact with multiple source code files in real time. Instead of generating reports or complexity results, the File-Saver functionality helps users save their uploaded files locally after editing or interacting with them, preserving their changes. This ensures a smooth workflow for users who need to manage multiple files while working collaboratively.

When users upload multiple source code files to the system, they can edit, review, or interact with them in real time. To facilitate saving these files back to the local system with the changes applied, File-Saver is employed to download the modified files. This provides a seamless experience, allowing users to easily manage their source code during collaborative sessions.

3) Code Execution with Piston API

For dynamic code analysis and execution, the system uses the Piston API [19]. The Piston API is an open-source multi-language code execution engine that allows the execution of code in various languages.

When the user uploads or inputs code into the system, the backend sends the code to the Piston API for execution and

retrieves results like output, error messages, or performance metrics (e.g., time taken to execute).

4) Complexity Calculation with Custom Java API

For dynamic code analysis and complexity metric calculations, the system employs a custom-built Java API designed to compute various metrics such as Time Complexity, Cyclomatic Complexity, Lines of Code (LOC), and Maintainability Index. This Java API was developed in-house to complement existing solutions like the Piston API, which, while effective for executing code in multiple languages, does not provide built-in functionality for calculating code complexity metrics.

When users upload or input their code into the system, the backend forwards the code to the custom Java API for analysis. The API runs the necessary algorithms to compute the required complexity metrics and returns results like time complexity, error messages, and other performance metrics. This integration ensures precise calculations, which are critical for both real-time collaboration and individual analysis workflows.

The Java API is built using Java Spring Boot and offers high performance and flexibility. It can handle multiple programming languages while maintaining accuracy in complexity calculations. The key metrics calculated by the API include:

- Lines of Code (LOC): This metric counts the total lines in the code, focusing on executable statements and omitting comments and white spaces.
- Halstead Metrics: These metrics help analyze the overall complexity and effort required for understanding and maintaining the code.
- Nesting Depth: This metric evaluates the depth of loops or conditional statements, which can indicate higher complexity when deep nesting is present.
- Maintainability Index: This index is calculated based on several factors, including the above metrics, to assess the overall maintainability of the code.

By integrating this custom Java API, the system offers accurate and efficient complexity analysis that goes beyond simple code execution, providing developers with actionable insights to improve code quality and maintainability.

5) Version Control and Automatic Updates

The system integrates with Git to track code changes and calculate complexity metrics based on the latest version of the codebase. Whenever new changes are pushed or committed to the repository, the system triggers a re-calculation of key metrics like Lines of Code (LOC), Cyclomatic Complexity, and Maintainability Index. This ensures that the metrics stay up-to-date with the evolving codebase, and users can view these changes directly from their dashboards.

The backend monitors the repository for changes and automatically recalculates the code complexity metrics. Users can access reports that show how the code's complexity evolves over time, which is useful for project management and planning.

The integration of various technologies such as Socket.io for real-time collaboration, File-Saver for easy downloading of source code files, the Piston API for executing code across multiple programming languages, and CodeMirror for an interactive code editor collectively forms a robust and efficient system for measuring and analyzing code complexity. This synergy not only enhances user experience but also fosters a collaborative environment where developers can work together seamlessly, regardless of their physical locations.

Socket.io enables instantaneous communication between users, allowing them to see each other's changes in real time, discuss code issues, and provide feedback on complexity metrics as they are being analyzed. This feature significantly accelerates the review process and minimizes the chances of miscommunication that often arise in traditional code review settings.

File-Saver facilitates the easy export and downloading of files, empowering users to preserve their edited code and analysis results locally. This functionality ensures that developers can maintain a clear version history and access their work without dependency on continuous internet connectivity. Being able to save and manage files directly enhances productivity and encourages users to experiment and iterate without fear of losing their changes.

Meanwhile, the Piston API allows for dynamic code execution, enabling the application to run code snippets in various programming languages. This capability provides immediate feedback on code behavior, which is essential for assessing performance and identifying potential complexity-related issues. By integrating execution results with complexity metrics, developers gain a holistic view of their code's quality and maintainability.

The inclusion of CodeMirror as the primary code editor enriches the user interface with features like syntax highlighting, code folding, and autocompletion. This interactivity not only makes coding more efficient but also encourages best practices by helping users adhere to coding standards through visual cues.

Furthermore, the system's integration with version control systems like Git ensures that code metrics are continuously updated in alignment with the codebase's evolution. This dynamic updating allows teams to track changes effectively, making informed decisions throughout the software development lifecycle. By linking complexity metrics to specific code versions, the system helps identify how changes affect code quality over time.

In summary, this combination of advanced technologies creates a seamless and efficient environment for developers. The application supports continuous improvement in code quality through real-time collaboration, easy file management, dynamic execution feedback, and an interactive coding experience, ultimately leading to better maintainability and reduced technical debt.

Overall system functionalities of the CCMM are illustrated in Figure 2.

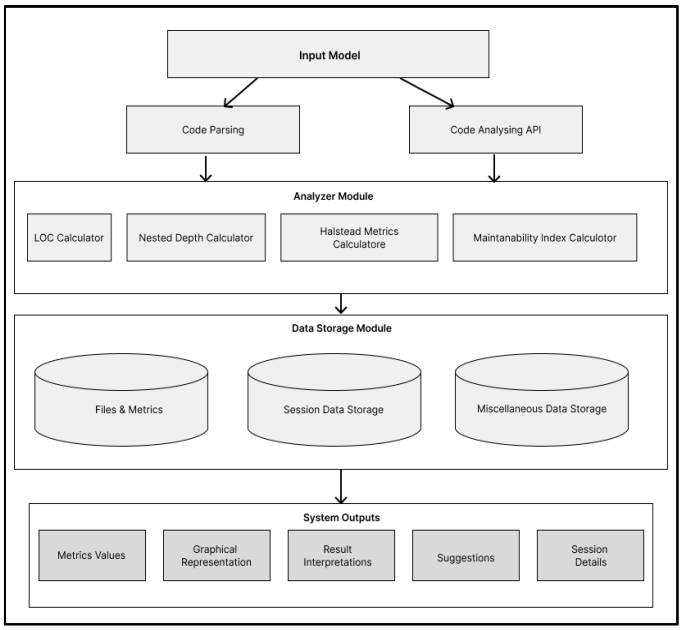

Figure 2. Higher Level View Of CCMM

IV. RESULTS

We successfully managed to implement CCMM. Not only it provides a basis for project code analysis, but also provides very user friendly and easy navigate interface to satisfy its users. Some of the functionalities that help users, performed by CCMM are demonstrated below. It provides an easy navigate buttons, easy access text area to paste your code, user friendly view of provided code and very clear results in every page of CCMM.

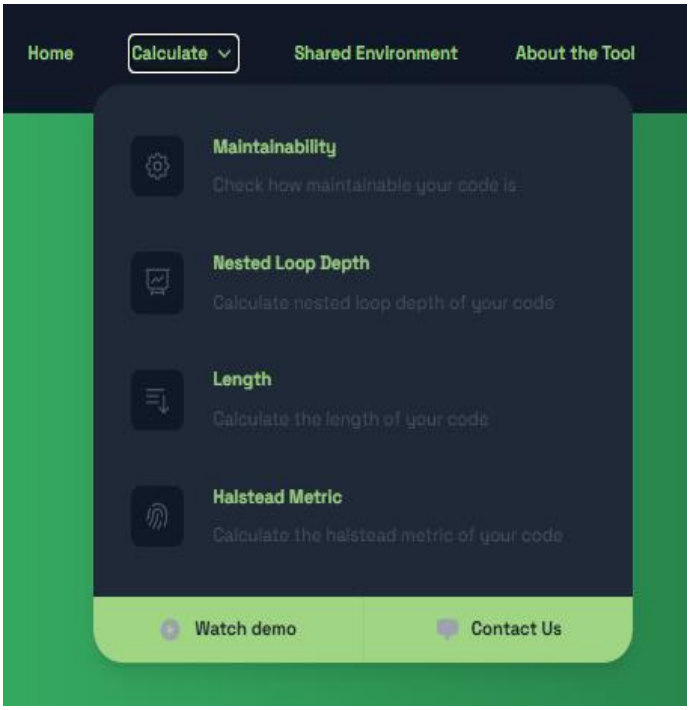

Figure 3. Navigation

In addition to its intuitive navigation and user-friendly interface, the Code Complexity Measuring Machine (CCMM) ensures that the complexity metrics are clearly defined in a way that users of all levels can easily understand. Each metric, such as Lines of Code (LOC), Nested Loop Depth, Halstead Complexity Metrics, and Maintainability Index, is accompanied by concise explanations, allowing users to interpret the results effectively. To enhance clarity further, CCMM uses a color-coded system where different colors represent various levels of code quality and complexity. Green indicate well-optimized code, while red highlights sections requiring attention. This visual distinction helps users quickly assess their code's status without delving deep into the numbers. The results are displayed with exceptional clarity, using well-organized text-based summaries, ensuring that users can effortlessly access and understand the analysis outcomes across every page of the application.

Figure 4. Code Maintainability Checker

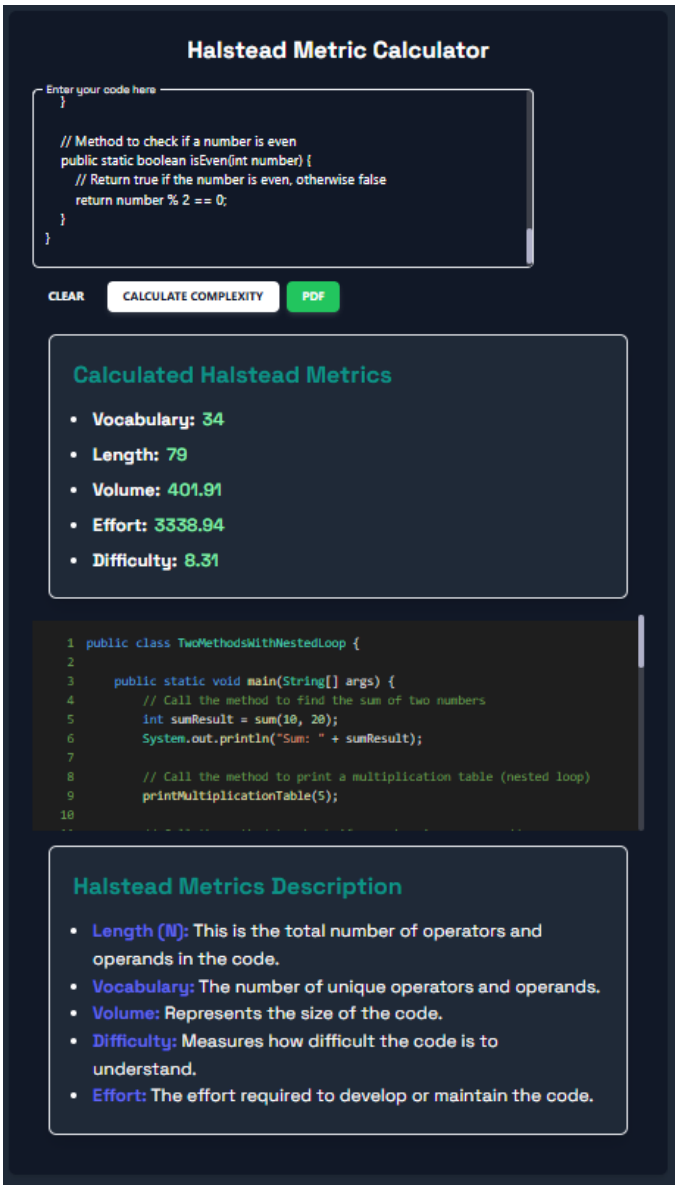

Figure 5. Halstead Metric Calculator

The UI for the collaborative application features a sidebar for navigation on the left, providing easy access to the project's file directory and other essential sections. The live file directory within this sidebar allows users to see real-time updates, with indicators like timestamps or dots next to files being actively worked on. Users' avatars are shown next to the files they are editing, offering a clear view of who is working on what. This intuitive sidebar navigation simplifies project management and file access during collaboration.

Integrated directly into the workspace, the chat window in the bottom-right allows for real-time communication, supporting both text and code snippets. This makes it easy for team members to discuss and share code without leaving the coding environment. Alongside the chat, a code execution panel enables users to run code in real-time, displaying outputs, errors, and logs in a terminal-like view, helping teams receive instant feedback on code functionality.

Within the main code editor, the UI supports multiple user interactions with each user's cursor and selections color-coded and labeled with their names. Real-time editing occurs seamlessly, with all collaborators seeing changes immediately as they happen. Additionally, a real-time metrics panel dynamically updates important coding metrics such as cyclomatic complexity and lines of code as the code evolves. These live metrics ensure that the entire team can monitor the quality and complexity of the project as they work together. Overall, the application offers a smooth, real-time collaborative experience with a sidebar-driven navigation for enhanced usability.

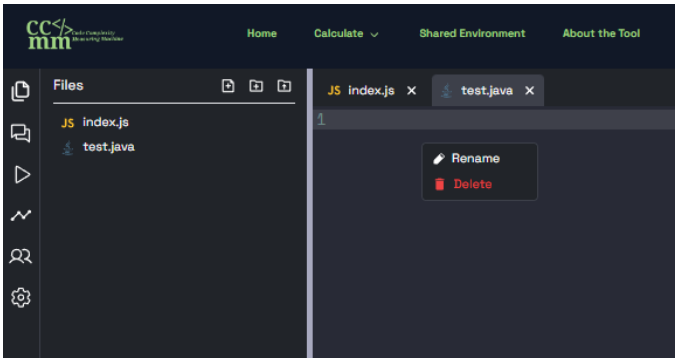

Figure 6. Shared Environment: Files

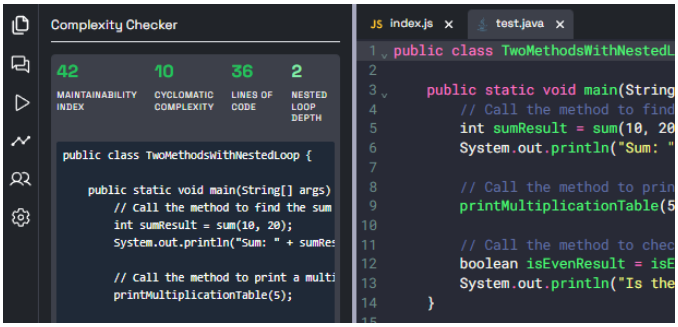

Figure 7. Real time Complexity Measurements

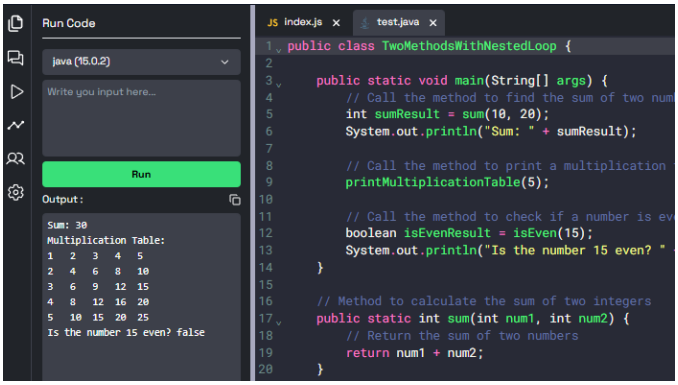

Figure 8. Code Execution

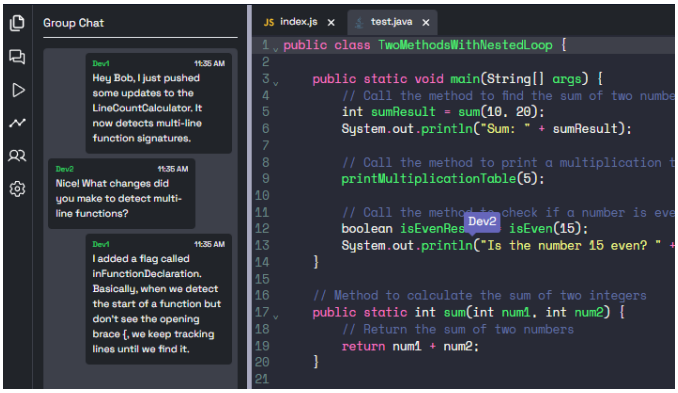

Figure 9. Chat and Realtime Collaborative Coding

These features work together to create a highly interactive and collaborative coding experience, allowing developers to streamline their workflow and focus on enhancing code quality. The integration of real-time collaboration with team members to work simultaneously on the same codebase, sharing insights, adding comments, and discussing issues without any delays. This real-time interaction ensures that feedback is immediate, fostering an environment where problems can be resolved quickly, and ideas can be shared openly. As a result, the collaborative process becomes more efficient, reducing the time spent on traditional code reviews and facilitating a smoother transition from development to deployment.

V. Discussions

The Code Complexity Measuring Machine (CCMM) has proven to be a powerful tool for assessing and improving the quality of software projects. By offering comprehensive complexity metrics like Lines of Code (LOC), Cyclomatic Complexity, Halstead Metrics, and the Maintainability Index, CCMM allows developers to gain valuable insights into their codebase's structural and logical intricacies. These metrics not only help in identifying potential problem areas but also guide developers in making informed decisions about refactoring, optimizing, and maintaining their code. With this detailed understanding of the code's complexity, developers can better manage technical debt and improve the overall stability and performance of their software.

One of the key strengths of CCMM is its focus on user experience. The design of a user-friendly interface with easy-to-navigate buttons and text areas simplifies the process of code input and analysis. Users can quickly paste their code, initiate analysis, and view results without needing extensive training or technical expertise. This ease of use significantly lowers the barrier to entry, making the tool accessible to both novice programmers and experienced developers. The integration of color-coded metrics further enhances user comprehension, allowing for quick visual identification of code areas that require optimization. For example, colors like green, yellow, and red signify different states of code quality, from well-optimized to high complexity or potential error-prone areas, making it easier for developers to focus on sections that need immediate attention.

The inclusion of real-time collaboration features, enabled by **Socket.io**, adds significant value to the system, transforming CCMM from a static analysis tool into a dynamic, interactive platform. This feature supports multiple users working together on the same project, making it an ideal choice for teams working in agile or distributed environments. The ability to see changes and comments in real-time not only speeds up the review process but also ensures that complexity-related issues are addressed collaboratively, fostering a culture of continuous improvement. This aspect of CCMM aligns well with modern development practices where collaboration and communication are critical to the success of software projects. It enables developers to conduct live discussions and peer reviews directly within the tool, leading to quicker identification of issues and faster resolution.

Furthermore, the custom-built Java API used in CCMM provides precise calculations for key metrics like Nesting Depth, Maintainability Index, and Halstead Metrics, making it possible to analyze various dimensions of code complexity with a high degree of accuracy. This feature is especially important when dealing with large codebases where even small inefficiencies can lead to significant performance issues. By using a tailored solution that focuses on these detailed metrics, CCMM provides a more specialized and in-depth analysis compared to generalized tools. This helps teams not only to maintain high code quality but also to plan future development phases more effectively by identifying complex modules that might require extra testing or refactoring.

Despite its many strengths, there are some areas where CCMM could be further enhanced. For example, while the Piston API allows for code execution across multiple languages, its role in complexity analysis could be expanded to include more detailed performance metrics like execution time profiling and memory usage statistics. Such features would allow developers to see not only how complex their code is but also how it performs in real-time, providing a more holistic view of code efficiency. Additionally, the integration of more advanced visualizations, such as heatmaps for code hotspots or interactive dependency graphs, could offer deeper insights into complex codebases. Such enhancements could help developers better understand the relationships between different parts of the code, leading to more effective optimizations. This would be particularly beneficial for teams working on large-scale systems where understanding module interactions is critical to maintaining stability.

Another potential improvement could involve expanding the scope of CCMM's collaborative features. Adding capabilities such as a history of changes or version comparison directly within the tool could further aid in tracking progress over time. These features would enable users to see how their code's complexity has evolved throughout the project lifecycle, allowing for retrospective analysis and better understanding of the impact of refactoring efforts. This could also be beneficial in educational settings, where instructors could use CCMM to track the progress of students as they learn to write more efficient and maintainable code.

Overall, the implementation of CCMM has achieved its primary goals, providing a robust framework for code analysis while maintaining a focus on user satisfaction. The combination of precise complexity metrics, intuitive design, and real-time

collaborative capabilities makes CCMM a valuable asset for development teams aiming to improve code quality and maintainability. As the complexity of software projects continues to grow, tools like CCMM will play a crucial role in helping teams manage technical debt, optimize performance, and ensure long-term project success. Additionally, with further improvements, CCMM could become a benchmark tool for not only development teams but also educational institutions, helping to build a foundation of best practices in coding, collaboration, and complexity management.

VI. Conclusion

The Code Complexity Measuring Machine (CCMM) application provides a comprehensive solution for analyzing, evaluating, and managing the complexity of software code, making it an essential tool for modern software development. By integrating multiple complexity metrics such as Lines of Code (LOC)**,** Cyclomatic Complexity**,** Halstead Metrics, Nesting Depth, and the Maintainability Index, the system empowers developers to gain a thorough understanding of the quality and maintainability of their codebase. These metrics offer insights into different aspects of the code, from the density and intricacy of logic to the ease with which code can be understood, tested, and modified. This holistic approach helps developers address both immediate code issues and long-term maintainability, facilitating higher code quality throughout the development lifecycle.

The application goes beyond traditional static code analysis by offering real-time collaboration features and dynamic file handling, which are crucial for software development teams operating in distributed or agile environments. The ability to collaboratively review and refine code in real-time helps teams catch potential issues early, reducing the need for extensive reworks later in the development process. With React as the front-end framework, the application offers a responsive and user-friendly interface, enabling developers to interact seamlessly with the system. Socket.io ensures that changes are propagated instantly among team members, keeping everyone in sync, while CodeMirror provides an interactive coding experience with syntax highlighting and other helpful features for editing code.

The system's backend, powered by a custom Java API built with Spring Boot, is designed to fill the gaps left by existing code execution tools like the Piston API. While the Piston API facilitates multi-language code execution, it lacks native support for complexity metric analysis. The custom API addresses this by providing robust and precise calculations for critical metrics, such as LOC, Halstead Metrics, and maintainability indices. By offering these detailed outputs, the application allows developers to assess the complexity of their codebase from multiple perspectives, enabling them to identify bottlenecks, understand how different parts of the code interact, and make informed decisions regarding optimization and refactoring. This deeper level of analysis supports teams in producing code that is not only functional but also maintainable and scalable.

A major strength of CCMM lies in its real-time collaboration capabilities, which significantly enhance team productivity. The **Socket.io** integration allows multiple users to simultaneously edit, comment on, and review code, enabling teams to discuss potential complexity issues as they emerge. This level of interactivity is particularly valuable in scenarios where quick feedback is needed, such as during code reviews or pair programming sessions. The **CodeMirror** editor complements this by providing a familiar coding environment that supports multiple programming languages, making it easier for team members to navigate and modify code. By catching issues early through collaborative review, CCMM helps teams maintain a steady pace without compromising code quality.

File-Saver plays a crucial role in file management, enabling users to save and export their code files directly from the browser. This feature allows developers to keep a local copy of their progress, preserving changes and enabling easy rollbacks if needed. This ability to manage source code files efficiently ensures that developers can maintain an organized workflow, even when handling complex projects with multiple files. Additionally, the integration of File-Saver facilitates seamless interaction with source files during collaborative sessions, allowing users to export their work in a consistent format, ensuring compatibility with other tools and platforms used by the team.

The application's custom Java API further enhances the workflow by performing real-time calculations of complexity metrics, such as Nesting Depth, which measures the depth of nested loops and conditionals, and Halstead Metrics, which analyze the complexity of operations and operands. By providing immediate feedback on how these metrics change as code is edited, CCMM enables a continuous improvement process. Developers can see the direct impact of their refactoring efforts, making it easier to achieve the desired balance between performance and maintainability. This continuous feedback loop encourages a culture of ongoing refinement and adherence to best coding practices, ensuring that the codebase remains maintainable and efficient as it evolves.

Beyond individual projects, CCMM also serves as a valuable tool for larger teams and organizations. Its ability to integrate with version control systems like **Git** means that complexity metrics can be updated automatically as code changes are committed, allowing teams to monitor the quality of their codebase over time. This feature is especially useful for tracking the effectiveness of refactoring efforts or assessing the impact of new features on the overall complexity of the project. By linking complexity data directly to version control, CCMM provides teams with a historical view of their codebase's evolution, enabling data-driven decision-making and better long-term planning.

In summary, the Code Complexity Measuring Machine (CCMM) is a fully-featured platform that not only aids developers in analyzing their code but also fosters a collaborative and continuous improvement environment. Its ability to provide precise complexity metric calculations, combined with real-time collaboration and robust file management, addresses the growing needs of modern development teams. CCMM supports efficient, high-quality code production, reducing technical debt, improving maintainability, and ensuring long-term software quality. As

software systems continue to increase in size and complexity, tools like CCMM are indispensable for maintaining the balance between innovation and stability, making it an invaluable asset for any development project.

VII. References

[1] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering,* Vols. SE-2, no. 4, pp. 308-320, Dec. 1976.

[2] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," *Proc. of International Conf. on Software Maintenance,* pp. 337-344, 1992.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[4] N. Fenton and S. L. Pfleeger, A Rigorous and Practical Approach, PWS Publishing Co., 1998.

[5] S. Weis and D. Lenk, "Collaborative real-time editing of web applications," *Proc. of ACM Symposium on User Interface Software and Technology (UIST),* pp. 205-212, 2011.

[6] M. H. Halstead, "Elements of Software Science," *Operating and Programming Systems Series,* vol. 7, 1977.

[7] D. Kafura and S. Henry, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering,* Vols. SE-7, no. 5, pp. 510-518, Sept. 1981.

[8] S. H. Kan, Metrics and Models in Software Quality Engineering, 2nd ed., Boston, MA: Addison-Wesley, 2002.

[9] M. H. Halstead, Elements of Software Science, New York, NY: Elsevier, 1977.

[10] Mikejo5000, "Code metrics - Maintainability index range and meaning - Visual Studio (Windows)," learn.microsoft.com. https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022, *learn.microsoft.com.*.

[11] "What is Cyclomatic Complexity? Definition Guide & Examples," *www.sonarsource.com. https://www.sonarsource.com/learn/cyclomatic-complexity/.*

[12] "Verifysoft → Halstead Metrics," *Verifysoft.com, 2024. https://www.verifysoft.com/en_halstead_metrics#:~:text=Halstead (accessed Sep. 17, 2024.*

[13] N. Weibel, A. Fouse, S. Emmenegger, and E. Hutchins, "Let's Get Physical: Spatialized Interaction in a Paper-Digital Collaborative Workspace," *Proc. of the ACM International Conf. on Interactive Tabletops and Surfaces,* p. 95–104, 2011.

[14] M. Storey, F. Figueira Filho, and L.-T. Cheng, "The Impact of Social Media on Software Development Practices and Tools," *Proc. of the 2014 International Conf. on Software Engineering,* pp. 24-35, 2014.

[15] A. Bacchelli and C. Bird, "Expectations, Outcomes, and Challenges of Modern Code Review," *Proc. of the International Conf. on Software Engineering (ICSE),* pp. 712-721, 2013.

[16] P. Rigby, D. M. German, and M. Storey, "Open Source Software Peer Review Practices: A Case Study of the Apache Server," *Proc. of the 30th International Conf. on Software Engineering (ICSE),* pp. 541-550., 2008,.

[17] A. Bosu, M. Greiler, and C. Bird, "Characteristics of Useful Code Reviews: An Empirical Study at Microsoft," *Proc. of the 12th Working Conf. on Mining Software Repositories (MSR),* pp. 146-156, 2015.

[18] C. F. Kemerer, "Software Complexity and Software Maintenance: A Survey of Empirical Research," *Annals of Software Engineering,* vol. 1, no. 1, pp. 1-22, 1995.

[19] Piston, "Piston API V2 Documentation," [Online]. Available: https://piston.readthedocs.io/en/latest/api-v2/.