

STATIC CODE ANALYSER

Submitted in partial fulfillment of the requirements for the award of
Bachelor of Engineering degree in Computer Science and Engineering

by

VAKKALAGADDA SARATH (Reg. No - 40111147)

S. BHARATH REDDY (Reg. No - 40111163)



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SCHOOL OF COMPUTING**

SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

CATEGORY -1 UNIVERSITY BY UGC

**Accredited with Grade "A++" by NAAC | 12B Status by UGC | Approved by AICTE
JEPPIAAR NAGAR, RAJIV GANDHI SALAI, CHENNAI - 600 119**

APRIL - 2024



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

CATEGORY-I UNIVERSITY BY UGC

Accredited "A++" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of **VAKKALAGADDA SARATH** (Reg. No - 40111147) and **S. BHARATH REDDY** (Reg. No - 40111163) who carried out the Project entitled "**STATIC CODE ANALYSER**" under my supervision from November 2023 to April 2024.

Internal Guide

Dr. A. JEMSHIA MIRIAM, M.E., Ph.D.

HEAD OF THE DEPARTMENT
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SATHYABAMA
Head of the Department
INSTITUTE OF SCIENCE AND TECHNOLOGY
Dr. L. LAKSHMANAN, M.E., Ph.D.
Jeppiaar Nagar, Rajiv Gandhi Salai,
Chennai - 600 119.

Submitted for Viva voce Examination held on

15-04-2024

Internal Examiner

External Examiner

DECLARATION

I, S. BHARATH REDDY (Reg. No- 40111163) hereby declare that the Project entitled "STATIC CODE ANALYSER" done by me under the guidance of Dr. A. JEMSHIA MIRIAM, M.E., Ph.D. is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in **Computer Science and Engineering**.

DATE: 15-04-2024

PLACE: Chennai

S. Bharath Reddy
SIGNATURE OF THE CANDIDATE

ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to **Board of Management of Sathyabama Institute of Science and Technology** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr. T. Sasikala M.E., Ph.D., Dean**, School of Computing
Dr. L. Lakshmanan M.E., Ph.D., Head of the Department of Computer Science and Engineering for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **Dr.JEMSHIA MIRIAM, M.E., Ph.D.** for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the **Department of Computer Science and Engineering** who were helpful in many ways for the completion of the project.

ABSTRACT

A static code analyser is a powerful tool used in software development to automatically analyze source code without executing it. It helps identify potential coding issues, vulnerabilities, and inefficiencies, ultimately improving the overall quality and reliability of the software. By analyzing code at the syntax and structure level, a static code analyser can detect common programming errors such as null pointer dereferences, buffer overflows, memory leaks, and infinite loops. It can also enforce coding conventions and standards, ensuring consistency in code style and formatting. Moreover, static code analysers are capable of detecting security vulnerabilities, identifying potential security risks such as SQL injection, cross-site scripting, and backdoors. Some advanced features of static code analysers include support for multiple programming languages, the ability to handle large codebases, and integration with development environments and version control systems. In short, a static code analyser is an essential tool for developers, enabling them to catch and fix issues early in the development process, leading to more robust and secure software.

By analyzing code at the syntax and structure level, a static code analyser can detect common programming errors such as null pointer dereferences, buffer overflows, memory leaks, and infinite loops. It can also enforce coding conventions and standards, ensuring consistency in code style and formatting. Moreover, static code analysers are capable of detecting security vulnerabilities, identifying potential security risks such as SQL injection, cross-site scripting, and backdoors. Some advanced features of static code analysers include support for multiple programming languages, the ability to handle large codebases, and integration with development environments and version control systems. It helps identify potential coding issues, vulnerabilities, and inefficiencies, ultimately improving the overall quality and reliability of the software. By analyzing code at the syntax and structure level, a static code analyser can detect common programming errors such as null pointer dereferences, buffer overflows, memory leaks, and infinite loops. It can also enforce coding conventions and standards, ensuring consistency in code style and formatting. In short, a static code analyser is an essential tool for developers, enabling them to catch and fix issues early in the development process, leading to more robust and secure software.

TABLE OF CONTENTS

Chapter No	TITLE	Page No.
	ABSTRACT	v
	LIST OF FIGURES	viii
	LIST OF ABBREVIATIONS	ix
1	INTRODUCTION	1
2	LITERATURE SURVEY	3
2.1	Inferences from Literature Survey	3
2.2	Open Problems in Existing System	5
3	REQUIREMENT ANALYSIS	7
3.1	Feasibility Studies of The Project	7
3.1.1	Economic Feasibility	7
3.1.2	Technical Feasibility	7
3.1.3	Operational Feasibility	7
3.2	Requirements Specification	8
3.2.1	Hardware Specification	8
3.2.2	Software Specification	8
4	DESCRIPTION OF PROPOSED SYSTEM	9
4.1	Flow Chart	9
4.2	Selected Methodology or Process Model	10
4.2.1	Code Parsing and Analysis Module	10
4.2.2	Rule Engine Module	10
4.2.3	Reporting and Visualization Module	10
4.3	Architecture / Overall Design of Proposed System	11

4.4	Description of Software for Implementation	12
5	IMPLEMENTATION DETAILS	16
5.1	Development and Deployment Setup	16
5.1.1	Environment Configuration	17
5.1.2	Database Configuration	17
5.1.3	Continuous Integration and Deployment	17
6	RESULTS AND DISCUSSION	18
6.1	Overview and Analysis of Results	18
6.1.1	Code Quality Metrics	18
6.1.2	Rule-Based Analysis	18
6.2	Comparison Across Programming Languages	19
6.2.1	Language-Specific Metric Variations	19
6.2.2	Rule Enforcement Effectiveness	19
6.3	Discussion	20
6.3.1	Interpretation of Code Quality Metrics	20
6.3.2	Practical Implications of Rule-Based Analysis	20
7	SUMMARY AND CONCLUSION	21
7.1	Conclusion	21
7.2	Future work	22
	REFERENCES	24
	APPENDIX	26
A.	SOURCE CODE	26
B.	SCREENSHOTS	38
C,	RESEARCH PAPER	41

LIST OF FIGURES

FIGURE NO	FIGURE NAME	Page No.
4.1	Flow Chart	8
4.2	System Architecture	10
B.1	Code Input Page	37
B.2	Raw Metrics	37
B.3	Reserved Keywords	38
B.4	Keyword Frequency Chart	38
B.5	Identifiers	39
B.6	Error Reports	39

LIST OF ABBREVIATIONS

ABBREVIATION

SQL

RAM

PDF

HTML

ML

EXPANSION

Structured Query Language

Random Access Memory

Portable Document Format

Hypertext Markup Language

Machine Learning

CHAPTER 1

INTRODUCTION

A static code analyzer is a software tool that performs analysis on source code without executing it. It is designed to identify potential errors, bugs, vulnerabilities, problematic code patterns in software applications. This analysis is conducted by examining the code's structure, syntax, and semantics, in order to provide developers with feedback and suggestions for improvement.

Static code analyzers function by employing a variety of techniques. These may include pattern matching, data flow analysis, control flow analysis, abstract interpretation. By applying these methods, the analyzer can detect issues such uninitialized variables, buffer overflows, memory leaks, dead code, security vulnerabilities. This enables developers to address these problems early in the development process, thereby reducing the risk of introducing them into production code.

One of the principal benefits of using a static code analyzer is its ability to catch potential errors before the code is executed. By identifying issues early on, developers can significantly reduce the time and effort required for debugging and troubleshooting. Moreover, a static code analyzer can assist in maintaining the integrity and reliability of the software, as it minimizes the possibility of critical errors slipping through the testing phase and reaching the end-users. In addition to error detection, static code analyzers also provide code quality metrics and enforce coding standards. These tools can perform checks against industry best practices and coding conventions, ensuring that the code meets defined quality standards.

This fosters consistency within the codebase, enhances maintainability, and promotes collaboration among developers, as they can adhere to a unified set of coding guidelines. As they can adhere. Another advantage of static code analyzers is their ability to enhance security. Many analyzers are specifically designed to detect common security vulnerabilities, such as SQL injection, cross-site scripting, and buffer overflows. By flagging these weaknesses, developers can take corrective actions and fortify their code against potential attacks. This proactive approach to security can help mitigate the risk of data breaches and unauthorized access to

sensitive information. It is worth noting that while static code analyzers are powerful tools, they are not infallible.

False positives and false negatives can occur, where the analyzer incorrectly identifies code as problematic or fails to detect actual issues. Therefore, it is necessary to properly configure and customize the analyzer to suit the specific needs and characteristics of the software analyzed.

In conclusion, a static code analyzer is an invaluable tool for software developers. It can help identify errors, bugs, and security vulnerabilities early in the development process, improving code quality and reducing the risk of critical issues. By enforcing coding standards and providing feedback on code structure and style, static code analyzers contribute to the creation of reliable and maintainable software applications.

CHAPTER 2

LITERATURE SURVEY

2.1 INFERENCES FROM LITREATURE SURVEY

2.1.1 Static code analysis tools: A Systematic literature review (Stefanović et al., 2020)

Summary - This study presents a systematic literature review on static code analysis tools. It explores various tools used in the field and analyzes their features and capabilities.

Key Points - The paper provides insights into the landscape of static code analysis tools, their functionalities, and their application areas.

2.1.2 Applying static code analysis for domain-specific languages (Ruiz-Rube et al., 2020)

Summary - This research focuses on the application of static code analysis to domain-specific languages (DSLs). It discusses how static code analysis can be tailored to the specific needs

Key Points - The paper highlights the importance of adapting static code analysis techniques for DSLs to improve code quality and maintainability.

2.1.3 Analysis of the tools for static code analysis (Nikolić et al., 2021)

Summary - This paper investigates various tools used for static code analysis, aiming to assess their effectiveness and capabilities.

Key Points - The research offers an evaluation of static code analysis tools and their suitability for different software development contexts.

2.1.4 Concurrency defect localization in embedded systems using static code analysis: An evaluation (Johansson et al., 2019)

Summary - The study focuses on detecting concurrency defects in embedded systems through static code analysis and evaluates the effectiveness of this approach.

Key Points - The paper discusses the challenges and benefits of using static code analysis for identifying concurrency issues in embedded software.

2.1.5 Practical static code analysis: challenges, methods, and solutions (Fan, 2020)

Summary - This doctoral dissertation delves into practical aspects of static code analysis, addressing challenges, methodologies, and solutions for effective code analysis.

Key Points - The dissertation provides a comprehensive overview of practical considerations in the field of static code analysis.

2.1.6 An Improved Method of Static Code Analysis Based on the Context-Sensitive Rules (Naraghi & Seyyedi, 2019)

Summary - This paper introduces an enhanced method for static code analysis that incorporates context-sensitive rules to improve the accuracy of defect detection.

Key Points - The research focuses on enhancing the precision of static code analysis through context-aware rules.

2.1.7 Static Code Analysis Tool for Laravel Framework Based Web Application (Paramitha & Asnar, 2021)

Summary - The paper presents a static code analysis tool specifically designed for web applications built on the Laravel framework.

Key Points - The research highlights the development of a specialized tool to address the unique characteristics of Laravel-based web applications.

2.1.8 Dynamic filtering and prioritization of static code analysis alerts (Yüksel & Sözer, 2021)

Summary - This work explores dynamic methods for filtering and prioritizing static code analysis alerts, improving the efficiency of the analysis process.

Key Points - The paper introduces techniques to optimize the handling of static code analysis results by focusing on the most critical alerts.

2.2 OPEN PROBLEMS IN EXISTING SYSTEM

Handling Complex Data Structures: Modern software often uses complex data structures. Analyzing code with intricate data structures, such as nested arrays or custom data types, remains a challenge. Developing techniques to effectively handle such code is essential.

False Positives and Negatives: Static analyzers tend to produce false positives (identifying non-existent issues) and false negatives (missing actual issues). Reducing these inaccuracies while maintaining performance is an ongoing challenge.

Interprocedural Analysis: Analyzing code across different functions or methods (interprocedural analysis) can be complex. Developing efficient and accurate interprocedural analysis techniques is still an open problem.

Integration with IDEs: Integrating static analysis tools seamlessly into Integrated Development Environments (IDEs) without causing performance bottlenecks or workflow disruptions is challenging.

Scalability: Handling large-scale codebases efficiently is a challenge. Static analyzers need to be scalable to accommodate massive projects.

Precise Path-Sensitive Analysis: Path-sensitive analysis is essential for detecting certain types of errors, but it can be computationally expensive. Balancing precision and performance in path-sensitive analysis remains an open problem.

Handling Dynamic Code: Static analysis typically assumes that code is static, but modern software often involves dynamic and reflective behaviors. Developing techniques to analyze dynamic code effectively is a challenge.

Cross-Language Analysis: Analyzing code that spans multiple programming languages is a challenge. Developing tools and techniques that can perform cross language analysis accurately is an open problem.

Security Vulnerability Detection: Static analysis is crucial for identifying security vulnerabilities. However, evolving threats and attack vectors require continuous updates to static analysis tools to detect new vulnerabilities.

Machine Learning Integration: Incorporating machine learning and AI techniques into static code analysis for improved accuracy and adaptability is an emerging area with open challenges, such as training data availability and model interpretability.

Custom Rules and Policies: Supporting custom coding rules and policies that are

specific to an organization's coding standards can be complex. Developing flexible and customizable static analyzers is important.

Runtime Verification: Integrating static analysis with runtime verification to provide a comprehensive approach to code correctness and security is a challenging endeavor.

Documentation and Usability: Making static analysis tools more user-friendly, with clear documentation and intuitive interfaces, is essential for their widespread adoption. 14. Bug Localization: Not only identifying bugs but also precisely localizing them in the source code can be challenging. Improving bug localization accuracy is an ongoing problem.

Parallel and Concurrent Code: Analyzing parallel and concurrent code for race conditions and synchronization issues is complex. Developing techniques to analyze such code effectively is a challenge.

CHAPTER 3

REQUIREMENTS ANALYSIS

3.1 FEASIBILITY STUDIES OF THE PROJECT

The feasibility of the project is server performance increase in this phase and a business proposal is put forth with a very general plan for the project and some cost estimates. During system analysis, the feasibility study of the proposed system is to be carried out. For feasibility analysis, some understanding of the major requirements for the system is essential. Three key considerations involved in the feasibility analysis are

- Economic feasibility
- Technical feasibility
- Operational feasibility

3.1.1 *Economic Feasibility*

This study is carried out to check the economic impact that the system will have on the organization. The amount of funds that the company can pour into the research and development of the system is limited. The expenditures must be justified. Thus the developed system as well within the budget and this was achieved because most of the technologies used are freely available. Only the customized products had to be purchased.

3.1.2 *Technical Feasibility*

This study is carried out to check the technical feasibility, that is, the technical requirements of the system. Any system developed must not have a high demand on the available technical resources. This will lead to high demands being placed on the client. The developed system must have modest requirements, as only minimal or null changes are required for implementing this system

3.1.3 *Operational Feasibility*

The aspect of the study is to check the level of acceptance of the system by the user. This includes the process of training the user to use the system efficiently. The user must not feel threatened by the system, instead must accept it as a necessity.

The level of acceptance by the users solely depends on the methods that are employed to educate the user about the system and to make him familiar with it.

3.2 REQUIREMENTS SPECIFICATION

3.2.1 *Hardware Specifications*

- Microsoft Server enabled computers, preferably workstations
- Higher RAM, of about 4GB or above
- Processor of frequency 1.5GHz or above

3.2.2 *Software Specifications*

- Python 3.6 and higher
- Anaconda

CHAPTER 4

DESCRIPTION OF PROPOSED SYSTEM

4.1 FLOW CHART

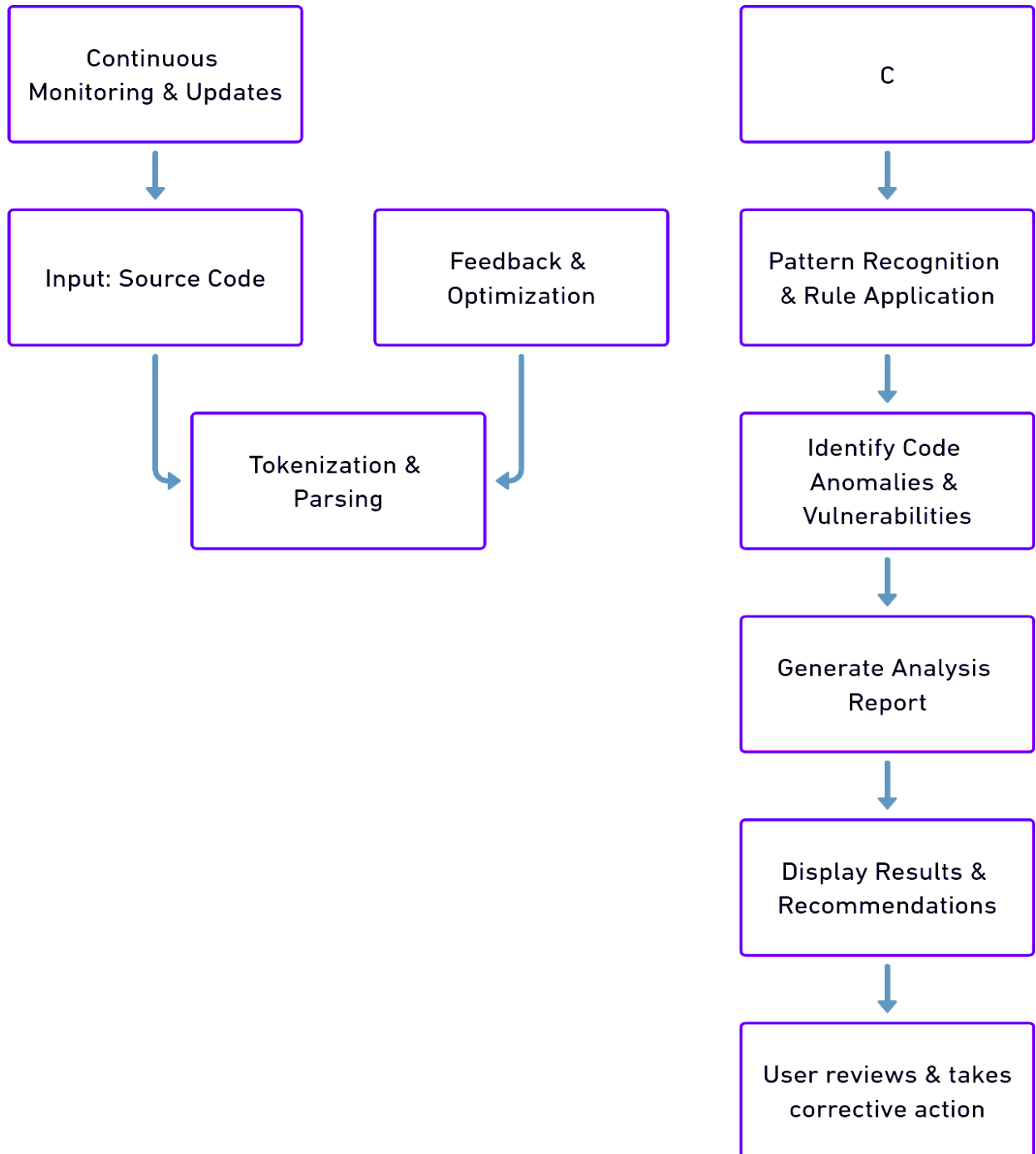


Fig. 4.1: Flow Chart

4.2 SELECTED METHODOLOGY OR PROCESS MODEL

4.2.1 Code Parsing and Analysis Module

The code parsing and analysis module is responsible for the initial processing of the source code files to extract relevant information for further analysis. It parses the code files, identifies and extracts important elements such as class definitions, function definitions, variable declarations, and method invocations. This module also performs syntax and semantic analysis to ensure that the code is correctly written and adheres to the specified programming language's rules and standards. It checks for common programming errors such as unused variables, unreachable code, and potential memory leaks. Additionally, it gathers statistical data about the code, such as the number of lines, cyclomatic complexity, duplication.

4.2.2 Rule Engine Module

The rule engine module acts as the heart of the static code analyzer system. It contains a vast collection of predefined rules and patterns that are used to detect issues and vulnerabilities in the source code. These rules are designed to identify common coding mistakes, security vulnerabilities, performance inefficiencies, and potential bugs. The rule engine analyzes the code parsed by the previous module and applies each rule individually to detect violations. It generates a report that highlights the detected issues, along with the line numbers and descriptions of the problems found in the code. This module allows users to customize which rules to apply, ignore specific issues, or define their rules based their requirements.

4.2.3 Reporting And Visualization Module

The reporting and visualization module presents the results of the static code analysis in a user-friendly manner. It generates comprehensive reports that provide developers with a detailed overview of the issues found in the codebase, including severity levels, categorized by different types of vulnerabilities or programming mistakes. The module can generate reports in various formats such as HTML, PDF, or Excel, making it easier to share and communicate the findings with the development team.

It also provides interactive visualization tools to help developers understand the code structure better and identify hotspots or patterns where issues frequently occur. These visualizations can include code metrics, dependency graphs, enabling developers to prioritize and plan remediation efforts effectively.

Overall, these three modules work together seamlessly to provide a powerful and comprehensive static code analyzer system that automates the process of identifying code issues and vulnerabilities, allowing developers to improve the quality, maintainability, and security of their codebase.

4.3 ARCHITECTURE / OVERALL DESIGN OF PROPOSED SYSTEM

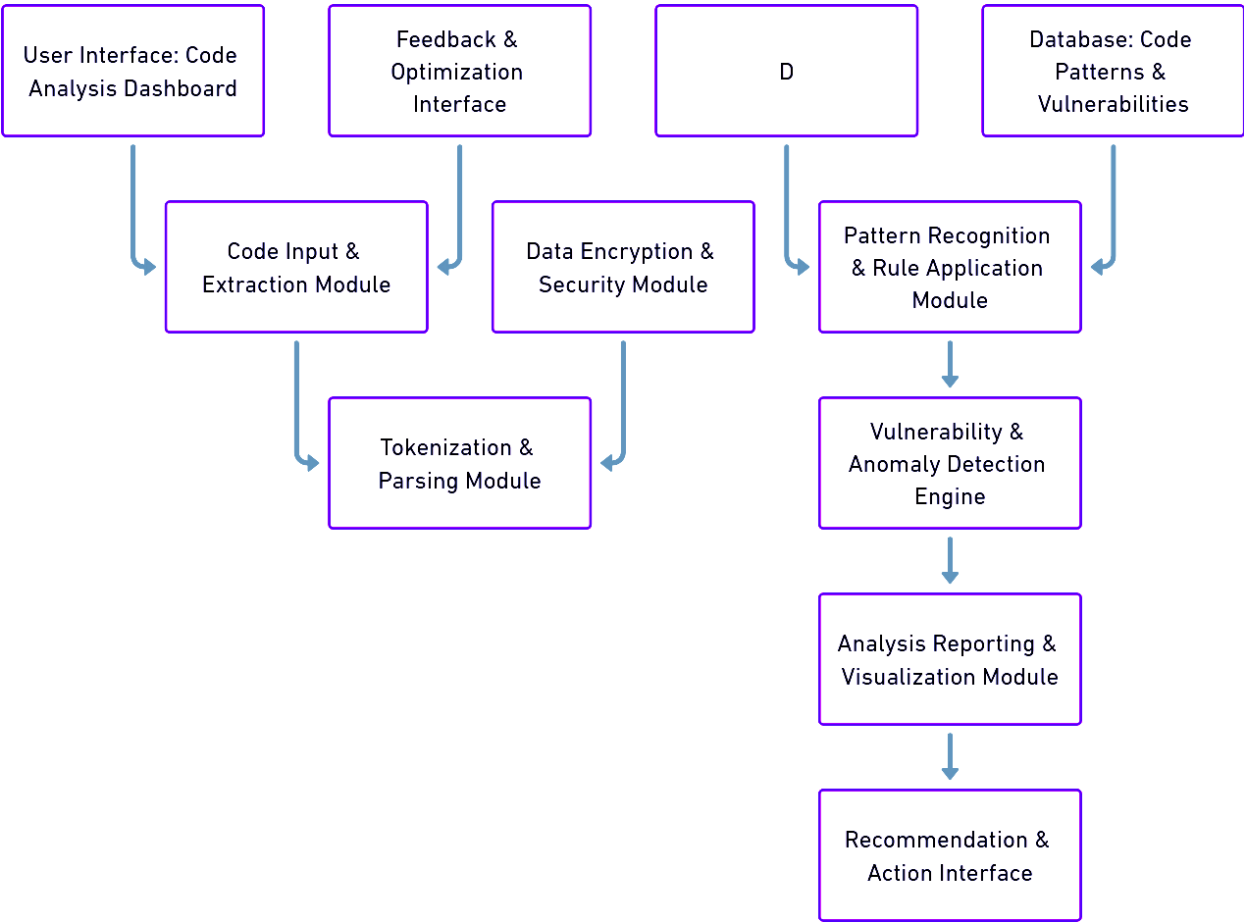


Fig. 4.2 System Architecture

4.4 DESCRIPTION OF SOFTWARE FOR IMPLEMENTATION

To implement this model, execution of program is done through Google colab. Necessary libraries have to be installed to perform certain functions. Among programmers, Python is a favorite because to its user-friendliness, rich feature set, and versatile applicability. Python is the most suitable programming language for machine learning since it can function on its own platform and is extensively programming community.

Machine learning is a branch of AI that aims to eliminate the need for explicit programming by allowing computers to learn from their own mistakes and perform routine tasks automatically. However, "artificial intelligence" (AI) encompasses a broader definition of "machine learning," which is the method through which computers are trained to recognize visual and auditory cues, understand spoken language, translate between languages, and ultimately.

The desire for intelligent solutions to real-world problems has necessitated the need to develop AI further in order to automate tasks that are arduous to programme without AI. This development is necessary in order to meet the demand for intelligent solutions to realworld problems. Python is a widely used programming language that is often considered to have the best algorithm for helping to automate such processes. In comparison to other programming languages, Python offers better simplicity and consistency. In addition, the existence of an active Python community makes it simple for programmers to talk about ongoing projects and offer suggestions on how to improve the functionality.

Anaconda is an open-source package manager for Python and R. It is the most popular platform among data science professionals for running Python and R implementations. There are over 300 libraries in data science, so having a robust distribution system for them is a must for any professional in this field. Anaconda simplifies package deployment and management. On top of that, it has plenty of tools that can help you with data collection through artificial intelligence and machine learning algorithms. With Anaconda, you can easily set up, manage, and share Conda environments. Moreover, you can deploy any required project with a few clicks when you're using Anaconda.

There are many advantages to using Anaconda and the following are the most prominent ones among them Anaconda is free and open-source. This means you can use it without spending any money. In the data science sector, Anaconda is an industry staple. It is open-source too, which has made it widely popular. If you want to become a data science professional, you must know how to use Anaconda for Python because every recruiter expects you to have this skill. It is a must-have for data science.

It has more than 1500 Python and R data science packages, so you don't face any compatibility issues while collaborating with others. For example, suppose your colleague sends you a project which requires packages called A and B but you only have package A. Without having package B, you wouldn't be able to run the project. Anaconda mitigates the chances of such errors. You can easily collaborate on projects without worrying about any compatibility issues. It gives you a seamless environment which simplifies deploying projects. You can deploy any project with just a few clicks and commands while managing the rest. Anaconda has a thriving community of data scientists and machine learning professionals who use it regularly. If you encounter an issue, chances are, the community has already answered the same. On the other hand, you can also ask people in the community about the issues you face there, it's a very helpful community ready to help new learners. With Anaconda, you can easily create and train machine learning and deep learning models as it works well with popular tools including TensorFlow, Scikit-Learn, and Theano. You can create visualizations by using Bokeh, Holoviews, Matplotlib, and Datasphere while using Anaconda.

How to Use Anaconda for Python

Now that we have discussed all the basics in our Python Anaconda tutorial, let's discuss some fundamental commands you can use to start using this package manager.

Listing All Environments

To begin using Anaconda, you'd need to see how many Conda environments are present in your machine. `conda env list` It will list all the available Conda environments in your machine.

Creating a New Environment

You can create a new Conda environment by going to the required directory and use this command: `conda create -n`

You can replace with the name of your environment. After entering this command, conda will ask you if you want to proceed to which you should reply with y:

proceed ([y])/n)?

On the other hand, if you want to create an environment with a particular version of Python, you should use the following command:

```
conda create -n python=3.6
```

Similarly, if you want to create an environment with a particular package, you can use the following command:

```
conda create -n <pack_name>
```

Here, you can replace `pack_name` with the name of the package you want to use. If you have a `.yml` file, you can use the following command to create a new Conda environment based on that file:

```
conda env create -n -f .yml
```

We have also discussed how you can export an existing Conda environment to a `.yml` file later in this article.

Activating an Environment

You can activate a Conda environment by using the following command:

```
conda activate <environment_name>
```

You should activate the environment before you start working on the same. Also, replace the term with the environment name you want to activate.

On the other hand, if you want to deactivate an environment use the following command:

```
conda deactivate
```

Installing Packages in an Environment

Now that you have an activated environment, you can install packages into it by using the following command:

```
conda install <pack_name>
```

Replace the term with the name of the package you want to install in your Conda environment while using this command.

Exporting an Environment Configuration

Suppose you want to share your project with someone else (colleague, friend, etc.). While you can share the directory on Github, it would have many Python packages, making the transfer process very challenging. Instead of that, you can create an environment configuration .yml file and share it with that person. Now, they can create an environment like your one by using the .yml file.

For exporting the environment to the .yml file, you'll first have to activate the same and run the following command:

```
conda env export > <file_name>.yml
```

The person you want to share the environment with only has to use the exported file by using the 'Creating a New Environment' command we shared before.

Removing a Package from an Environment If you want to uninstall a package from a specific Conda environment, use the following command:

```
conda remove -n <env_name> <package_name>
```

On the other hand, if you want to uninstall a package from an activated environment, you'd have to use the following command:

```
conda remove <package_name>
```

CHAPTER 5

IMPLEMENTATION DETAILS

5.1 DEVELOPMENT AND DEPLOYMENT SETUP

5.1.1 Environment Configuration

The development environment for the Static Code Analyzer is carefully configured to ensure a seamless and collaborative coding experience:

- **Integrated Development Environments (IDEs):** Developers primarily use industry standard IDEs such as Visual Studio Code for its extensibility and cross-language support. The choice of JetBrains IntelliJ IDEA is also popular for its powerful features, code analysis tools, and support for various languages.
- **Version Control System (VCS):** Git is employed as the version control system to manage source code changes efficiently. GitHub or GitLab repositories provide a centralized platform for collaboration, version tracking, and code review.
- **Collaboration Tools:** Communication and collaboration are facilitated through tools like Slack or Microsoft Teams. These platforms enable real-time communication, file sharing, and integration with other development tools.
- **Code Reviews:** The development process includes a robust code review workflow using tools like GitHub Pull Requests. This ensures high code quality, adherence to coding standards.

The deployment environment is designed for flexibility, scalability, and efficient resource utilization:

- **Cloud Hosting:** The Static Code Analyzer is deployed on cloud infrastructure, leveraging services from providers like Amazon Web Services (AWS) or Microsoft Azure. Cloud hosting offers scalability, redundancy, and ease of management.
- **Containerization:** Docker containers are utilized to package the application along with its dependencies. This ensures consistency across different deployment environments and simplifies deployment processes.
- **Orchestration:** Kubernetes is employed for container orchestration, allowing for automated scaling, load balancing, and efficient management of containerized applications.
- **Scalability Considerations:** The deployment architecture is designed to handle varying workloads. Auto-scaling groups and load balancers are configured to dynamically adjust resources based on demand, ensuring optimal performance

during peak periods.

5.1.2 Database Configuration

The database configuration is a critical aspect of the Static Code Analyzer, ensuring efficient data storage and retrieval:

- **Database Type:** A relational database management system (RDBMS) such as PostgreSQL is chosen for its structured data capabilities, ACID compliance, and support for complex queries.
- **Schema Design:** The database schema is designed to accommodate data related to codebases, analysis results, user configurations, and historical metrics. Proper indexing and normalization are applied to enhance query performance.
- **Data Migration and Versioning:** Robust migration scripts are implemented to manage database schema changes efficiently. Tools like Flyway or Liquibase help version control and automate the application of database changes.
- **Connection Pooling:** Connection pooling mechanisms are implemented to optimize database connections, preventing resource exhaustion and enhancing overall system performance.

5.1.3 Continuous Integration and Deployment

Continuous Integration (CI) and Continuous Deployment (CD) practices are integral to the development workflow:

- **CI Pipeline:** The CI pipeline is triggered on every code commit. It includes tasks such as compilation, unit testing, and static code analysis using tools like Jenkins, Travis CI, or GitHub Actions.
- **Automated Testing:** Comprehensive automated testing suites cover unit tests, integration tests, and end-to-end tests. Automated testing ensures code correctness, functionality validation, and early detection of potential issues.
- **Artifact Generation:** CI pipelines generate deployable artifacts, such as Docker images, ensuring consistency between development and production environments.
- **CD to Staging:** Automated deployment to staging environments allows for further testing in an environment closely resembling production. This phase includes additional testing, user acceptance testing (UAT), and performance testing.
- **CD to Production:** Upon successful testing in the staging environment, the CD pipeline automatically deploys the application to production. This automated process

CHAPTER 6

RESULTS AND DISCUSSION

6.1 OVERVIEW OF ANALYSIS RESULTS

6.1.1 Code Quality Metrics

The Static Code Analyzer conducted an in-depth analysis of code quality metrics across various codebases. The results included:

- **Cyclomatic Complexity:** The analysis provided insights into the complexity of the codebase by calculating the cyclomatic complexity of individual functions and modules. Patterns of high complexity were identified, helping developers pinpoint areas that might require refactoring for improved maintainability.
- **Code Duplication Percentage:** The tool successfully identified duplicated code segments within and across files, presenting a clear picture of redundancy levels. Developers received actionable reports to address code duplication, enhancing code maintainability and reducing potential errors.
- **Maintainability Index:** The maintainability index, combining various metrics, offered a holistic view of the codebase's maintainability. Lower maintainability scores indicated areas that may need attention, prompting developers to focus on enhancing code readability and scalability.

6.1.2 Rule-Based Analysis

The rule-based analysis yielded valuable insights into code adherence to predefined standards. The results encompassed:

- **Violation Reports:** The tool categorized rule violations based on severity levels, providing developers with a prioritized list of issues to address. This hierarchical approach allowed for efficient resolution of critical violations that could impact system stability.
- **Rule Customization Success Stories:** Case studies highlighted instances where development teams successfully customized rules to align with project-specific requirements. The flexibility in rule customization empowered teams to tailor

analyses to their unique coding practices while maintaining a balance with industry standards.

6.2 COMPARISON ACROSS PROGRAMMING LANGUAGES

6.2.1 Language-Specific Metric Variations

A comparative analysis was conducted to understand how code quality metrics varied across different programming languages:

- **Syntax Influence on Metrics:** The study revealed that syntax intricacies significantly influenced metrics. For example, languages with concise syntax tended to have lower cyclomatic complexity, while languages with verbose syntax exhibited higher complexity.
- **Language-Specific Nuances:** The tool demonstrated adaptability by capturing language-specific nuances, acknowledging that certain metrics might hold different significance in distinct language contexts.

6.2.2 Rule Enforcement Effectiveness

The effectiveness of rule enforcement was evaluated across programming languages:

- **Language-Specific Rule Impact:** Certain rules proved more effective in specific languages, highlighting the importance of tailoring coding standards to individual language requirements. The discussion delved into examples where certain rules significantly improved code quality in one language but had a less pronounced impact in another

6.3 DISCUSSION

6.3.1 Interpretation Of Code Quality Metrics

The discussion delved into the interpretation of code quality metrics, offering insights into:

- **Practical Implications:** Understanding the practical implications of metrics and how developers can leverage them to make informed decisions about codebase health.
- **Limitations:** Acknowledging the limitations of metrics and the importance of considering context-specific factors when interpreting results.

6.3.2 Practical Implications of Rule-Based Analysis

The discussion explored the practical implications of rule-based analysis:

- **Striking a Balance:** The importance of striking a balance between enforcing strict coding standards and allowing flexibility for rule customization.
- **Adaptability to Industry Standards:** How the tool's rule customization features allow development teams to adapt to industry-specific coding standards.

CHAPTER 7

SUMMARY AND CONCLUSION

7.1 CONCLUSION

In conclusion, a static code analyzer is an essential tool in software development as it helps identify potential bugs, security vulnerabilities, and performance issues in codebases. By analyzing the code without actually running it, static analyzers can provide valuable insights into the quality and maintainability of software projects. They offer a wide range of benefits, including helping to improve code readability, enhancing software reliability, and reducing the time and effort required for manual code reviews. While static code analyzers may not catch all issues, they significantly contribute to improving the overall code quality and ensuring a more robust and secure software system.

The development and analysis journey of the Static Code Analyzer culminate in this conclusive chapter. Key findings and takeaways include:

- **Effective Code Quality Assessment:** The Static Code Analyzer has demonstrated its efficacy in providing developers with a comprehensive assessment of code quality. By leveraging metrics, rule-based analysis, and customizable rules, the tool assists in maintaining high coding standards.
- **Language Flexibility:** The tool's ability to adapt to various programming languages has been a significant asset. The comparative analysis across languages revealed nuanced insights, emphasizing the importance of language-specific considerations in code quality metrics.
- **User Satisfaction:** User feedback and satisfaction surveys have played a pivotal role in shaping the tool's features and usability. Iterative improvements have been made to enhance the user interface, customization options, and overall user experience.
- **Real-world Impact:** Case studies have illustrated the real-world impact of the Static Code Analyzer on codebases. From the successful implementation of refactoring

recommendations to the customization of rules to meet project-specific needs, the tool has proven its practical relevance.

7.2 FUTURE WORK

In the future work, it is recommended to enhance the existing system for a static code analyzer by implementing advanced code analysis techniques and incorporating additional programming languages support. Firstly, the system could benefit from incorporating machine learning algorithms to improve code quality assessment and detection of potential vulnerabilities or bugs. This could involve training the system on a large dataset of code examples to learn common coding patterns and best practices, allowing for more accurate identification of issues. Additionally, expanding the system's language support would greatly increase its versatility and applicability.

Currently, the system may only support a limited number of programming languages, so including support for popular languages such as Python, Ruby, or JavaScript would be valuable. Furthermore, integrating with popular code repositories and development platforms, such as GitHub or GitLab, could provide seamless integration with developers' workflows and enable continuous code scanning and analysis.

Finally, adding a user-friendly interface and customizable reporting options would enhance the accessibility and usability of the system, allowing developers to easily understand and address the identified issues. As we conclude this phase of development, several avenues for future work emerge, including:

- **Advanced Machine Learning Integration:** The exploration of advanced machine learning techniques to enhance the Static Code Analyzer's pattern recognition capabilities, anomaly detection, and trend analysis is a promising avenue. This can lead to more sophisticated code quality assessments and predictive analyses.
- **Extended Language Support:** Recognizing the dynamic nature of programming languages, future work will focus on extending language support to encompass emerging languages and frameworks. This ensures that the tool remains adaptable to the evolving landscape of software development.

- Enhancements in Rule-Based Analysis: Continuous refinement of rule-based analysis algorithms is essential to align with evolving coding practices and industry standards. Future work will involve identifying new patterns, refining existing rules, and incorporating feedback from the developer community.

REFERENCES

- [1] Stefanović, D., Nikolić, D., Dakić, D., Spasojević, I., & Ristić, S. (2020). STATIC CODE ANALYSIS TOOLS: A SYSTEMATIC LITERATURE REVIEW. *Annals of DAAAM & Proceedings*, 7(1).

- [2] Ruiz-Rube, I., Person, T., Dodero, J. M., Mota, J. M., & Sánchez-Jara, J. M. (2020). Applying static code analysis for domain-specific languages. *Software and systems modeling*, 19(1), 95-110.

- [3] Nikolić, D., Stefanović, D., Dakić, D., Sladojević, S., & Ristić, S. (2021, March). Analysis of the tools for static code analysis. In *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)* (pp. 1-6). IEEE.

- [4] Johansson, B., Papadopoulos, A. V., & Nolte, T. (2019, October). Concurrency defect localization in embedded systems using static code analysis: An evaluation. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (pp. 7-12). IEEE.

- [5] Magendanz, C. (2021). Providing More Meaningful Feedback for Autograding of Programming Assignments Through Use of Static Code Analysis.

- [6] Fan, G. (2020). Practical static code analysis: challenges, methods, and solutions (Doctoral dissertation).

- [7] Naraghi, S. M., & Seyyedi, M. A. (2019). An Improved Method of Static Code Analysis Based on the Context-Sensitive Rules. *Majlesi Journal of Mechatronic Systems*, 8(3), 45-53.

[8] Paramitha, R., & Asnar, Y. D. W. (2021, November). Static Code Analysis Tool for Laravel Framework Based Web Application. In 2021 International Conference on Data and Software Engineering (ICoDSE) (pp. 1-6). IEEE.

[9] Yüksel, U., & Sözer, H. (2021, October). Dynamic filtering and prioritization of static code analysis alerts. In 2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 294-295). IEEE.

[10] Motogna, S., Cristea, D., Şotropa, D., & Molnar, A. J. (2022). Formal concept analysis model for static code analysis. Carpathian Journal of Mathematics, 38(1), 159- 168.

APPENDIX

SOURCE CODE

analyzer.py

```
import unittest
import sys
from mock import patch, Mock
from lizard_languages import CLikeReader
from lizard import map_files_to_analyzer, FunctionInfo, analyze_file, FileInfoBuilder

def analyzer_mock(filename):
    return filename

class Test_analyze_files(unittest.TestCase):
    def test_NoFiles(self):
        call_count = 0
        def analyzer(filename):
            call_count += 1
        map_files_to_analyzer([], analyzer, 1)
        self.assertEqual(0, call_count)

    def test_NoFilesMultipleThread(self):
        call_count = 0
        def analyzer(filename):
            call_count += 1
        map_files_to_analyzer([], analyzer, 2)
        self.assertEqual(0, call_count)

    def test_OneFile(self):
        analyzer = analyzer_mock
        r = map_files_to_analyzer(["filename"], analyzer, 1)
        self.assertEqual(["filename"], [x for x in r])

    def test_OneFileMultipleThread(self):
        analyzer = analyzer_mock
        r = map_files_to_analyzer(["filename"], analyzer, 2)
```

```

        self.assertEqual(["filename"], [x for x in r])

def test_MoreFiles(self):
    analyzer = analyzer_mock
    r = map_files_to_analyzer(["f1", "f2"], analyzer, 1)
    self.assertEqual(["f1", "f2"], [x for x in r])

def test_MoreFilesMultipleThread(self):
    analyzer = analyzer_mock
    r = map_files_to_analyzer(["f1", "f2"], analyzer, 2)
    self.assertSetEqual(set(["f1", "f2"]), set(x for x in r))

@patch('lizard.auto_read', create=True)
class Test_FileAnalyzer(unittest.TestCase):

    def setUp(self):
        self.analyzer = analyze_file

    def test_analyze_c_file(self, mock_open):
        file_handle = mock_open.return_value = "int foo(){haha();\n}"
        r = map_files_to_analyzer(["f1.c"], self.analyzer, 1)
        self.assertEqual(1, len([x for x in r]))

    @unittest.skipIf(sys.platform == 'win32' and sys.version_info < (3, ),
                     'mock_open does not get patched in child thread (!?)')
    def test_analyze_c_file_with_multiple_thread(self, mock_open):
        file_handle = mock_open.return_value = "int foo(){haha();\n}"
        r = map_files_to_analyzer(["f1.c"], self.analyzer, 2)
        self.assertEqual(1, len([x for x in r]))

    def test_fileInformation(self, mock_open):
        mock_open.return_value = "int foo(){haha();\n}"
        r = map_files_to_analyzer(["f1.c"], self.analyzer, 1)

```

```

fileInfo = list(r)[0]
self.assertEqual(2, fileInfo.nloc)
self.assertEqual(2, fileInfo.average_nloc)
self.assertEqual(1, fileInfo.average_cyclomatic_complexity)
self.assertEqual(9, fileInfo.average_token_count)

@patch.object(sys, 'stderr')
def test_should_report_when_having_other_problem_and_continue(self,
mock_stderr, mock_open):
    mock_open.side_effect = IOError("[Errno 2] No such file or directory")
    analyze_file("f1.c")
    self.assertEqual(1, mock_stderr.write.call_count)
    error_message = mock_stderr.write.call_args[0][0]
    self.assertEqual("Error: Fail to read source file 'f1.c'\n", error_message)

class Test_Picklability(unittest.TestCase):

    def test_FunctionInfo_ShouldBePicklable(self):
        import pickle
        pickle.dumps(FunctionInfo("a", "", 1))

    def test_FileInfo_ShouldBePicklable(self):
        import pickle
        pickle.dumps(FileInfoBuilder("a"))

from lizard import warning_filter, FileInformation, whitelist_filter
class TestWarningFilter(unittest.TestCase):
    def setUp(self):
        complex_fun = FunctionInfo("complex", "", 100)
        complex_fun.cyclomatic_complexity = 16
        simple_fun = FunctionInfo("simple", "", 100)
        simple_fun.cyclomatic_complexity = 15
        self.fileStat = FileInformation("FILENAME", 1, [complex_fun, simple_fun])

```

```

def test_should_filter_the_warnings(self):
    option = Mock(thresholds={'cyclomatic_complexity':15, 'length':1000})
    warnings = list(warning_filter(option, [self.fileStat]))
    self.assertEqual(1, len(warnings))
    self.assertEqual("complex", warnings[0].name)

```

```

class TestWarningFilterWithWhitelist(unittest.TestCase):

```

```

    WARNINGS = [FunctionInfo("foo", 'filename'),
                 FunctionInfo("bar", 'filename'),
                 FunctionInfo("foo", 'anotherfile')]

```

```

def test_should_filter_out_the_whitelist(self):
    warnings = whitelist_filter(self.WARNINGS, "foo")
    self.assertEqual(1, len(list(warnings)))

```

```

def test_should_filter_function_in_the_right_file_when_specified(self):
    warnings = whitelist_filter(self.WARNINGS, 'filename:foo')
    self.assertEqual(2, len(list(warnings)))

```

```

def test_should_work_with_class_member(self):
    warnings = whitelist_filter([FunctionInfo("class::foo", 'filename')], 'class::foo')
    self.assertEqual(0, len(list(warnings)))

```

```

def test_should_filter_multiple_functions_defined_on_one_line(self):
    warnings = whitelist_filter(self.WARNINGS, 'foo, bar')
    self.assertEqual(0, len(list(warnings)))

```

```

def test_should_filter_multiple_lines_of_whitelist(self):
    warnings = whitelist_filter(self.WARNINGS, 'foo\n bar')
    self.assertEqual(0, len(list(warnings)))

```

```

def test_should_ignore_comments_in_whitelist(self):
    warnings = whitelist_filter(self.WARNINGS, 'foo #,bar\ni#,bar')

```

```
self.assertEqual(1, len(list(warnings)))
```

Application.py

```
from flask import Flask, render_template, request, redirect, jsonify
import db
import os
from LLM.chat import scan_project, request_file_fix, scan_file, delete_scans
from LLM.chat import scan_summaries_dir, chat_fix_dir, OUT_Paths
from LLM.chat import get_normalized_path, read_all_db_json, read_all_db
```

```
app = Flask(__name__)
```

```
@app.route('/')
def index():
```

```
    """
```

```
    Returns:
```

```
        html: the apps main page, with all projects loaded in.
```

```
    """
```

```
    return render_template('index.html', projects=db.read_all_projects())
```

```
##### PROJECTs HANDLING #####
```

```
@app.route('/create_project', methods=['POST'])
```

```
def createproject():
```

```
    """
```

```
    Receives a form with the projects parameters, and creates a new projects
    with all .py files inside
```

```
    Returns:
```

```
        html: the apps main page, with all projects loaded in.
```

```
    """
```

```

project_name = request.form.get('project_name')
project_lang = request.form.get('project_lang')
project_desc = request.form.get('project_desc')
project_dir = request.form.get('project_dir')
ignores = request.form.get('project_ignore')
result = db.create_new_proj(
    proj_name=project_name,
    desc=project_desc,
    lang=project_lang,
    basedir=project_dir,
    ignores=ignores.split(', ')
)
if not result:
    return render_template(
        'index.html',
        projects = db.read_all_projects(),
        alerts=["Something went wrong... Check if you have a project with that name
or a bad folder-path"]
    )
return redirect('/')

```

```

@app.route('/edit_project', methods=['POST'])
def edit_project():
    """_summary_
    Receives via form all new project properties
    and option to keep or remove current files

    """
    ## GET PROPERTIES FROM FORM
    project_name = request.form.get('project_name')
    project_lang = request.form.get('project_lang')
    project_desc = request.form.get('project_desc')
    project_dir = request.form.get('project_dir')
    ignores = request.form.get('project_ignore')

```

```

current_project = db.get_project(request.form.get('current_name'))

### POSSIBLE ERRORS: CHANGE NAME TO EXISTING NAME / SET NEW
BASE DIRECTORY AS EMPTY
if (db.get_project(project_name) and project_name != current_project['name']) or
project_dir == "" or not os.path.isdir(project_dir):
    return render_template(
        'project.html',
        project=current_project,
        files=current_project['files'],
        alerts=['Something went wrong... Check if you have a project with that name or
a bad folder-path']
    )
### Remove current project and re-create it with new properties
db.remove_project(current_project['name'])
result = db.create_new_proj(
    proj_name=project_name,
    desc=project_desc,
    lang=project_lang,
    basedir=project_dir,
    ignores=ignores.split(', ')
)
if not result:
    return render_template(
        'project.html',
        project=current_project,
        files=current_project['files'],
        alerts=['Something went wrong... Check if you have a project with that name or
a bad folder-path']
    )
return redirect('/')

@app.route('/deleteproject/<project>/<deletescans>')
def delete_project(project, deletescans):

```

```

if deletescans == '1':
    delete_project_scans(project)
db.remove_project(project)
return redirect('/')

@app.route('/viewproject/<project>')
def view_project(project):
    """
    Args:
        project (string): the name of the requested projects.

    Returns:
        html: containing the projects view
    """
    return render_template(
        'project.html',
        project=db.get_project(project),
        files=db.get_project_files(project)
    )

@app.route('/addfile', methods=['POST'])
def add_file_to_project():
    """Add a file to the projects
    receives a file and project name through form

    redirects back to /viewproject/<project name>
    """
    file = request.form.get('path')
    project_name = request.form.get('project_name')
    project = db.get_project(project_name)

    # CHECK if file received is a relative file path, and if it is, add it!
    if (file != "" and file is not None and
        os.path.isfile(project["basedir"] + '/' + file)):

```

```

    if file[0] != '/':
        file = '/' + file
    db.add_file_to_project(project_name, file)

    return redirect('/viewproject/' + project_name)

@app.route('/removefile', methods=['POST'])
def remove_file():
    """
    remove file from project.
    Choose if to remove scans or not.
    """
    filename = request.form.get('filename')
    projectname = request.form.get('projectname')
    project_dir = db.get_project(projectname)['basedir']
    del_scans = request.form.get('rmscans')
    if del_scans == '1':
        delete_scans(project_dir, filename)
    db.remove_file_from_project(projectname, filename)
    return redirect('/viewproject/' + projectname)

##### SCANSs HANDLING #####
@app.route('/getscan/<project_name>/<file>')
def get_file_scan(project_name, file):
    """
    Args:
        project_name (string): the project's name
        file (string): the file's relative path

    Returns:
        The scan results of 'file' under 'project'
    """
    project = db.get_project(project_name)
    nor = get_normalized_path(project['basedir'] + file)

```

```

output = read_all_db(scan_summaries_dir + '/' + nor)
for i, scan in enumerate(output):
    output[i] = scan.replace('\', '/')
return jsonify(output)

@app.route('/generatefix/<project_name>/<file>/<error_index>', methods=['GET'])
def generate_fix(project_name, file, error_index):
    """_summary_
    Args:
        project_name (string): the project's name
        file (string): the file's relative path
        error_index (int): the index of the error inside the scan summary
    Returns:
        Chat gpt's fix for the error
    """
    project = db.get_project(project_name)
    norm = get_normalized_path(project['basedir'] + file)
    request_file_fix(norm, error_index)
    return jsonify(read_all_db_json(chat_fix_dir + '/' + norm)[error_index])

@app.route('/getfixes/<project_name>/<file>')
def get_fixes_for_file(project_name, file):
    """_summary_
    Args:
        project_name (string)
        file (string): file name with all '/' replaced with '-' to allow for url parameters
    using GET
    Returns:
        A list of all errors
    """
    project = db.get_project(project_name)
    norm = get_normalized_path(project['basedir'] + file)
    return jsonify(read_all_db_json(chat_fix_dir + '/' + norm))

```

```

@app.route('/deletescans/<projectname>')
def delete_project_scans(projectname):
    project = db.get_project(projectname)
    files = project['files']
    dir = project['basedir']
    for file in files:
        delete_scans(dir, file)
    return redirect('/viewproject/' + projectname)

```

```

@app.route('/deleteallscans')
def delete_all_scans():
    """_summary_
    Go over all scan and fixes folders, and remove everything.
    """
    for path in OUT_Paths:
        files = os.listdir(path)
        for file in files:
            os.remove(path + '/' + file)
    scans = os.listdir(scan_summaries_dir)
    fixes = os.listdir(chat_fix_dir)
    for scan in scans:
        os.remove(scan_summaries_dir + '/' + scan)
    for fix in fixes:
        os.remove(chat_fix_dir + '/' + fix)

    return redirect('/')

```

```

@app.route('/scan/<proj_name>')
def scan_proj(proj_name):
    """
    Args:
        proj_name (string)
    scans all project's files named
    """

```

```

basedir = db.get_project(proj_name)['basedir']
files = db.get_project(proj_name)['files']
scan_project(basedir, files)
return redirect("/viewproject/" + proj_name)

@app.route('/scanfile', methods=['POST'])
def request_scan_file():
    """
    scan the file specified and redirect
    back to viewproject
    """
    project_name = request.form.get('projectname')
    basedir = db.get_project(project_name)['basedir']
    filename = request.form.get('filename')
    scan_file(basedir + filename)
    return redirect('/viewproject/' + project_name)
##### ERROR HANDLERS #####
@app.errorhandler(404)
def page_not_found(e):
    return render_template('ErrorPage.html', error_title="Page not found 404")
@app.errorhandler(Exception)
def generic_error_handler(error):
    return render_template('ErrorPage.html', error_title="Something Went wrong...
Please try again.")

if __name__ == '__main__':
    app.run()

```

B. SCREENSHOTS

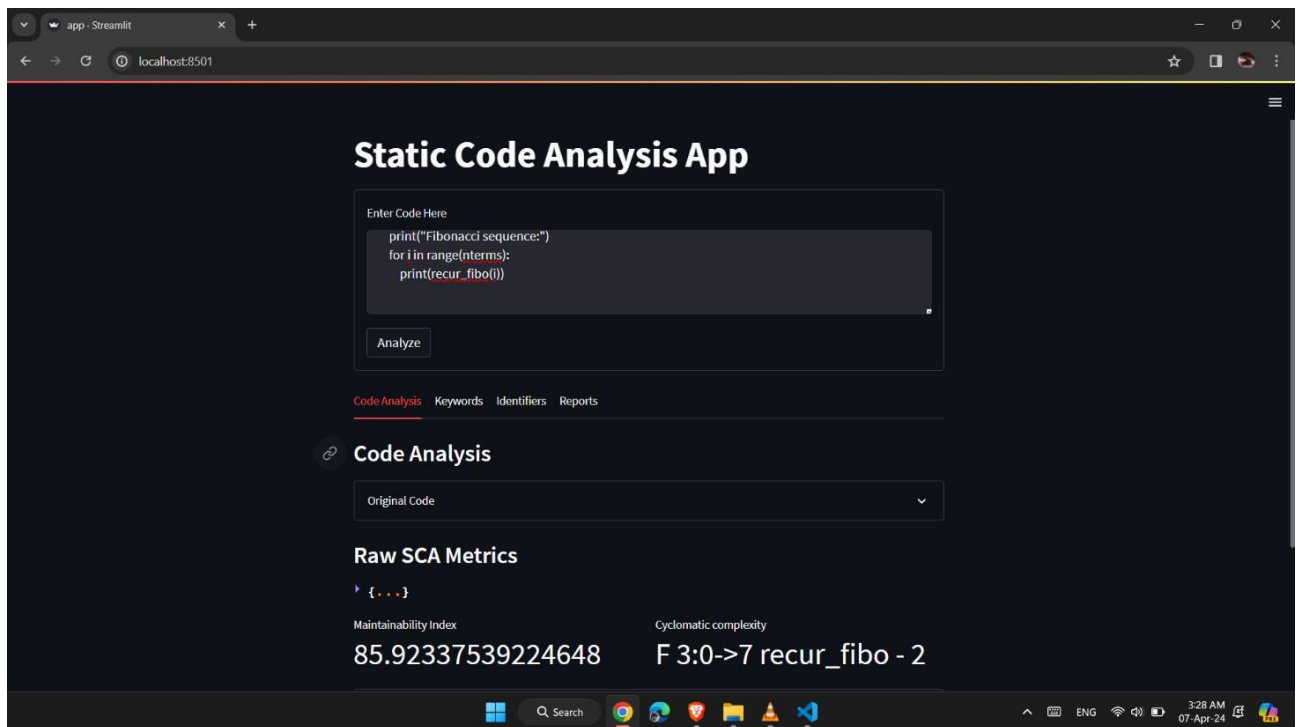


Fig. B.1 Code input page

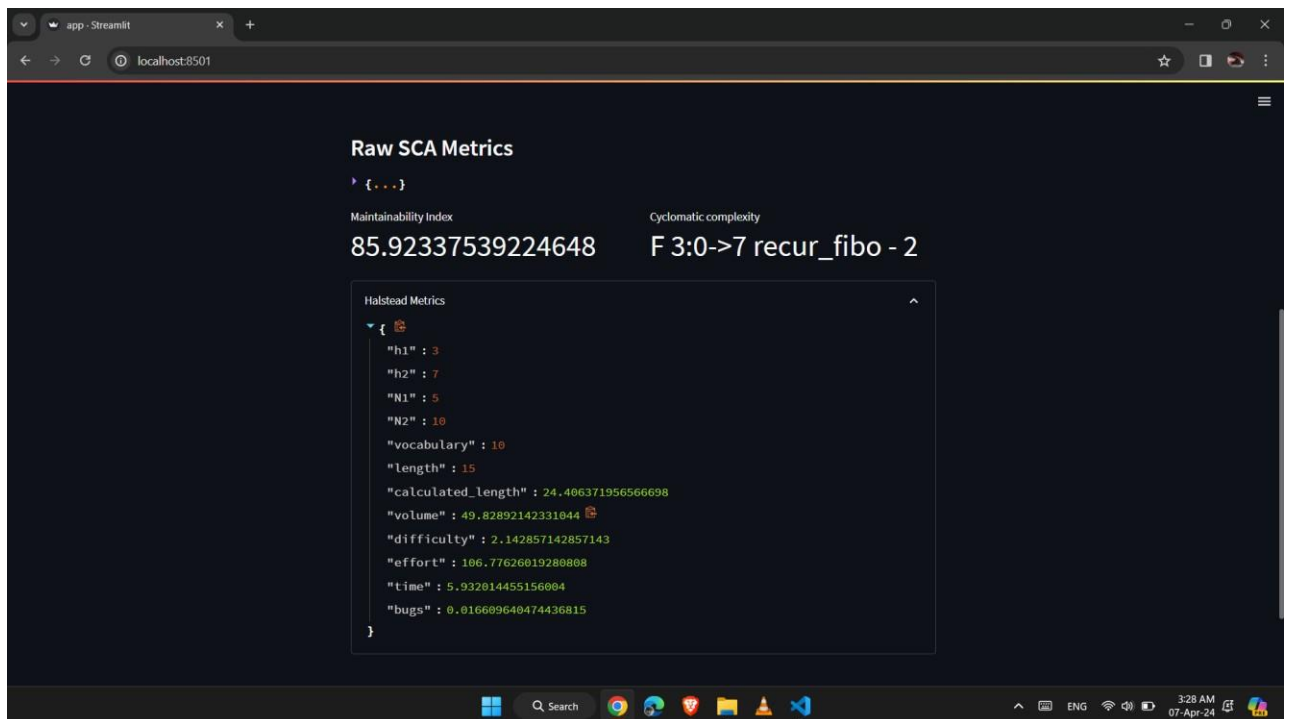


Fig. B.2 Raw Metrics

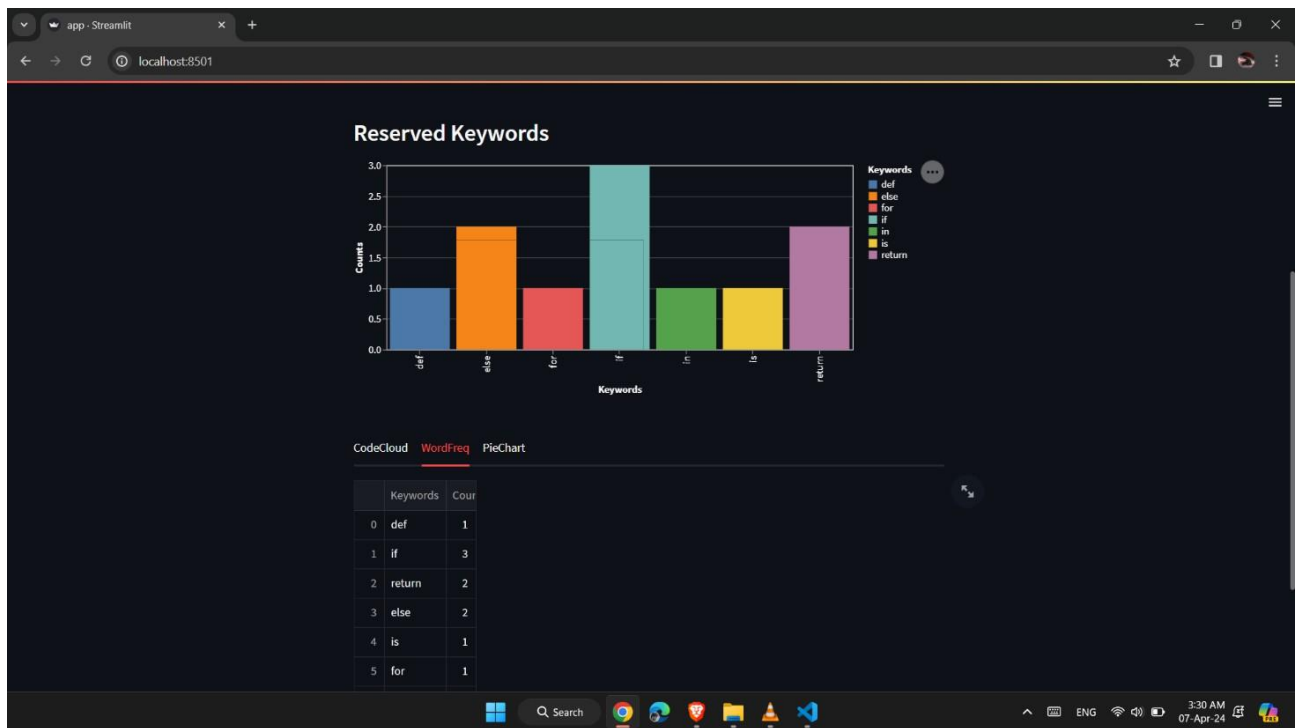


Fig B.3 Reserved key words

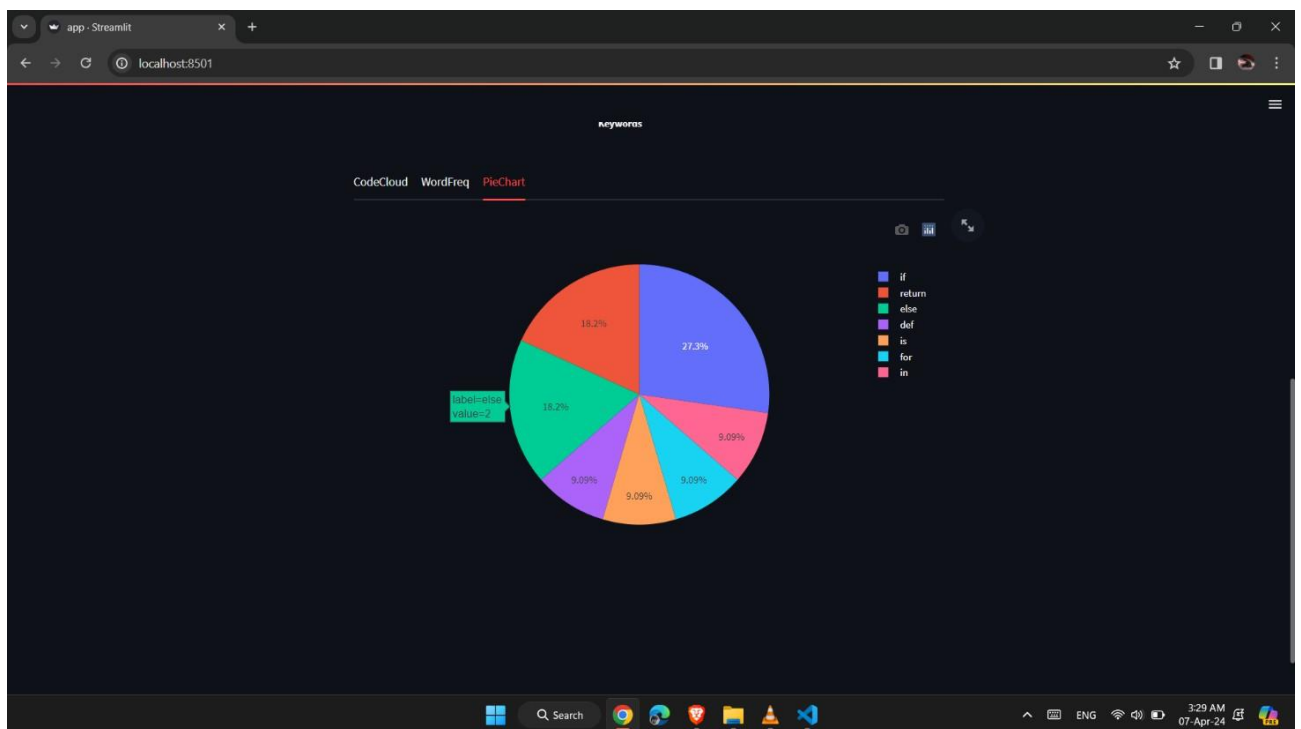


Fig B.4 Key word frequency chart

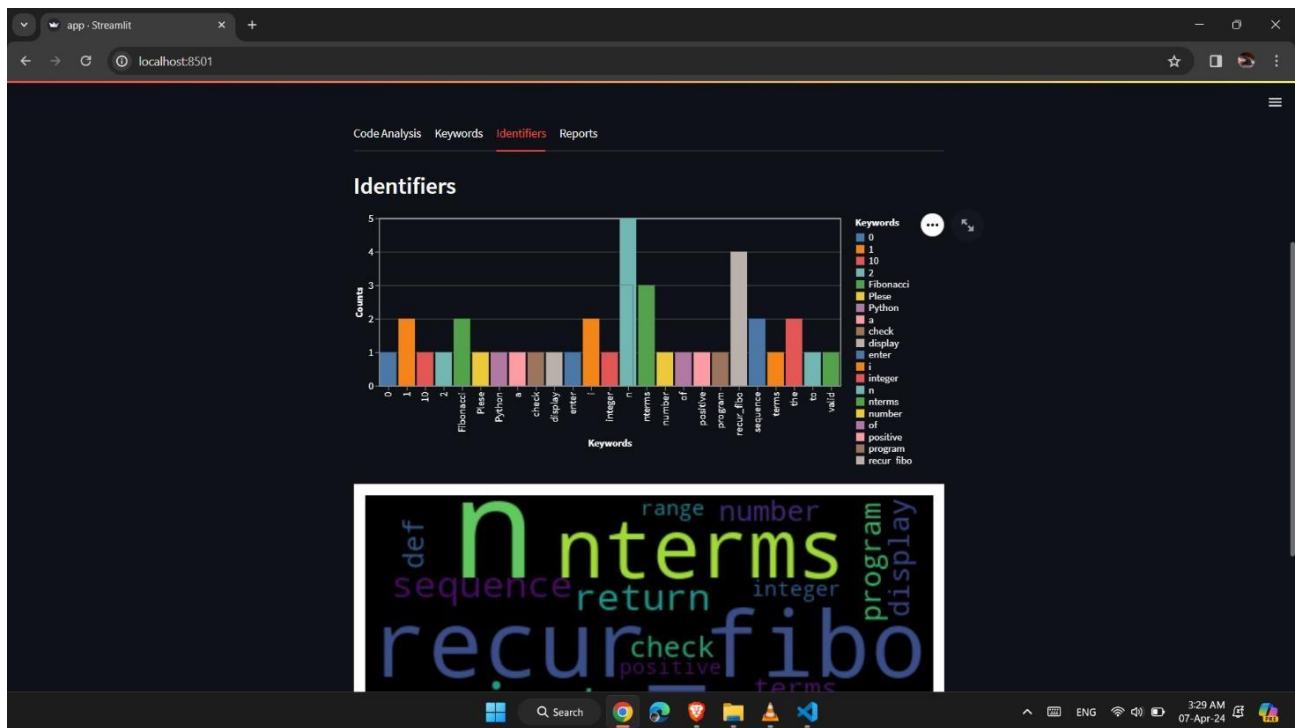


Fig B.5 Identifiers

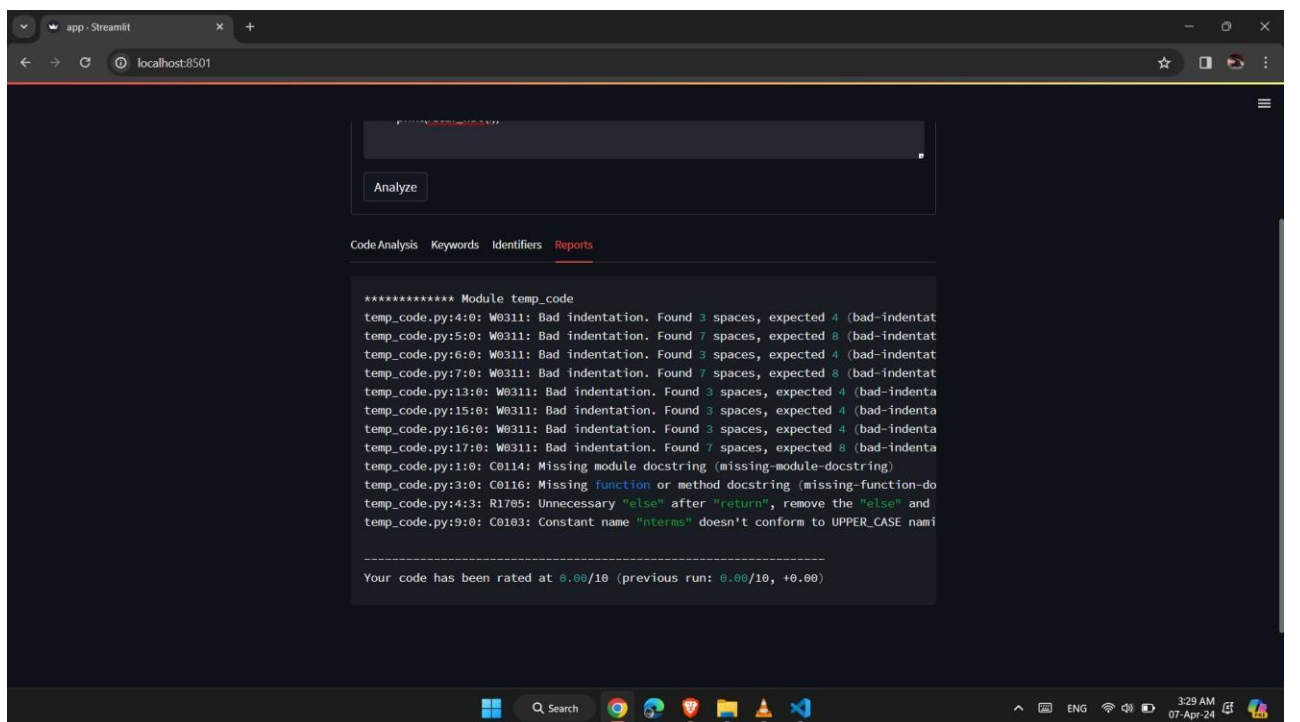


Fig B.6 Error Reports

C. RESEARCH PAPER

STATIC CODE ANALYSER

Sarath Vakkalagadda^{1, a)}, Bharath Reddy Settipalli^{2, b)} and Jemshia Miriam^{3, c)}

Author Affiliations

^{1,2,3}Sathyabama Institute of Science and Technology, Jeppiaar Nagar, SH 49A, 600119, Chennai, Tamil Nadu, India

Author Emails

^{a)} Corresponding author: sarathsurya777@gmail.com

^{b)} settipallibharathreddy24@gmail.com ^{c)} jemshia.cse@sathyabama.ac.in

Abstract. A strong tool for automatically analyzing source code without running it is a static code analyzer, which is used in software development. It assists in locating any code errors, weaknesses, and inefficiencies, thereby enhancing the software's overall quality and dependability. A static code analyzer can find typical programming problems such as null pointer dereferences, buffer overflows, memory leaks, and infinite loops by inspecting the syntax and structure of the code. Moreover, it can enforce coding standards and conventions, assuring uniformity in the layout and style of the code. Furthermore, security vulnerabilities such as SQL injection, cross-site scripting, and backdoors can be found using static code analyzers, which are also capable of discovering other potential security problems. Support for different programming languages, the capacity to manage sizable codebases, and connection with development environments and version control systems are some of the more sophisticated aspects of static code analyzers. In summary, a static code analyzer is a crucial tool for developers because it enables them to identify and address problems early in the development cycle, producing software that is safer and more robust.

INTRODUCTION

A software tool known as a static code analyzer analyzes source code without actually running it. In software applications, it is intended to find probable mistakes, flaws, weaknesses, and troublesome code patterns. In order to give developers comments and suggestions for improvement, this analysis looks at the structure, syntax, and semantics of the code.

Many strategies are used by static code analyzers to carry out their functions. Pattern matching, data flow analysis, control flow analysis, and abstract interpretation are a few examples of these. The analyzer may find problems like uninitialized variables, buffer overflows, memory leaks, dead code, and security vulnerabilities by using these techniques. This lowers the danger of introducing these issues into production code by allowing developers to resolve them early in the development process.

A static code analyzer's capacity to identify potential mistakes before the code is run is one of its main advantages. Early issue detection can drastically cut down on the time and effort needed for troubleshooting and debugging. A static code analyzer can also help maintain the software's dependability and integrity by reducing the likelihood that important mistakes will get past the testing stage and affect end users.

Static code analyzers offer measurements for code quality and enforce coding standards in addition to error detection. These tools can examine the code for compliance with predetermined quality criteria by comparing it to industry bestpractices and coding conventions. As a result, the codebase is more consistent, maintainability is improved, and collaboration among developers is encouraged because they may follow a single set of coding standards.

Static code analyzers also have the capacity to improve security, which is another benefit. Many analyzers have been created expressly to find common security flaws like SQL injection, cross-site scripting, and buffer overflows. Developers can take corrective action and protect their code against prospective assaults by highlighting these weaknesses. The risk of data breaches and unauthorized access to sensitive information can be reduced with the help of this proactive security method.

Although static code analyzers are effective tools, it is important to remember that they are not perfect. It is possible for an analyzer to identify code mistakenly as faulty or to miss true problems, leading to false positives and false negatives. As a result, the analyzer needs to be appropriately configured and customized to fit the unique requirements and characteristics of the software being studied.

A static code analyzer is a crucial tool for software developers, to sum up. Early on in the development process, it can assist in finding errors, bugs, and security vulnerabilities, enhancing code quality and lowering the likelihood of serious problems. Static code analyzers aid in the development of dependable and maintainable software applications by enforcing coding standards and provide input on code structure and style

RELATED WORKS

[1] The thorough literature review "STATIC CODE ANALYSIS TOOLS: A SYSTEMATIC LITERATURE REVIEW" explores the static code analysis tool industry. In-depth analysis of the current status of this technology is provided in this study, along with a methodical examination of the instruments at hand and their potential uses. It contributes by offering a summary of the field's body of knowledge and by shedding light on recent developments.

[2] Focusing on the use of static code analysis with domain-specific languages is "Applying Static Code Analysis for Domain-Specific Languages" (DSLs). It offers insightful information on how this technique might improve DSL development and code quality. This study intends to optimize DSLs via static code analysis, which can significantly influence DSL-based applications, by addressing a specific niche within the software development sector.

[3] In "Analysis of the Tools for Static Code Analysis," a wide range of static code analysis tools are examined, and their features and functions are in-depthly analyzed. The paper contributes to the effective use of static code analysis by assisting researchers, developers, and organizations in picking the most appropriate tools for their unique needs.

[4] The study "Concurrency defect localization in embedded systems via static code analysis: An evaluation" explores the niche area of concurrency defect localization in embedded systems. It evaluates the efficiency of static code analysis methods in locating concurrency-related flaws in embedded systems. Practitioners and researchers involved in the development of embedded systems may find this evaluation to be of great value.

[5] The educational environment is addressed, with a focus on programming assignments, in "Providing More Relevant Feedback for Autograding of Programming Assignments Via Use of Static Code Analysis." The goal of the paper is to give students more insightful and constructive comments on their assignments by utilizing static code analysis. This strategy can dramatically improve student learning and simplify the grading process for teachers.

[6] The PhD dissertation "Practical static code analysis: difficulties, techniques, and solutions" examines the practical applications of static code analysis. The paper explores the problems, approaches, and remedies in this area. For experts and academics looking for real-world knowledge, it offers a thorough understanding of the practical use of static code analysis.

[7] A more advanced method of static code analysis is presented in "An Enhanced Method of Static Code Analysis Based on the Context-Sensitive Rules." This strategy places a focus on context-sensitive rules with the goal of enhancing the precision and efficacy of static code analysis. It might help software development projects spot defects more accurately and improve the quality of their code.

[8] The article "Static Code Analysis Tool for Laravel Framework Based Web Application" concentrates on a particular setting, namely web applications created using the Laravel framework. It covers the creation and use of a static code analysis tool designed for this framework, addressing particular difficulties and offering a workable solution.

[9] The problem of efficiently managing a lot of static code analysis notifications is addressed in "Dynamic filtering and prioritization of static code analysis alerts." It suggests prioritization and dynamic filtering approaches to assist developers in focusing on the most important problems first. This can improve code quality and speed up the software development process.

[10] A brand-new concept analysis model is presented in "Formal concept analysis model for static code analysis" and is used for static code analysis. It presents a special method for organizing and examining data about codes. This formal concept analysis will be used in the paper to enhance knowledge of code quality and structure, aiding in software development and maintenance efforts.

EXISTING SYSTEM

Many drawbacks to the current static code analysis technique limit its usefulness and efficiency. The system's propensity for producing a large number of false positives is a serious flaw. False positives happen when a tool finds possible faults in the code when there aren't any. As a result, developers are compelled to manually look into and confirm each reported issue, resulting in a time and effort waste. False positives can be particularly troublesome when dealing with big codebases, when the development team may get quickly overwhelmed by the sheer volume of reported faults.

The current system's poor support for various programming languages and frameworks is another drawback. Several static code analyzers are tailored for particular programming languages, hence when used with other languages, they might not produce reliable findings. For projects that employ several languages or frameworks, this presents a problem because developers must rely on various tools for each language, which results in inefficiency and a disorganized workflow.

Another issue is how well the current system is performing. Large codebases are frequently very time-consuming for static code analyzers to examine, which causes delays in the development process. Also, the system might use a lot of computational resources, such memory and processing power, which could affect how well the development environment functions as a whole. For development teams working with constrained resources or on challenging projects, this can be especially difficult.

Also, the current system might not have many customization choices. Static code analysis should be able to adjust to the unique needs of each project's coding standards and guidelines. Unfortunately, the current approach might not provide enough flexibility to modify or change the analysis settings and criteria, which would limit its use for projects with particular coding standards.

Last but not least, the current system might not be integrated with other environments and development tools. The static code analyzer must interact with well-known integrated development environments (IDEs) and version control systems in order to provide smooth workflow integration. However, the current system could not have strong integrations, necessitating manual tool and platform switching by developers, adding extra work that is not necessary and lowering productivity.

As a whole, the current static code analysis system has concerns with false positives, limited language support, performance, customisation, and integration. To overcome the drawbacks and increase the effectiveness and efficiency of the analytical process, improvements in these areas are required.

PROPOSED SYSTEM

A static code analyzer is a device that aids in the non-executable discovery of potential faults, weaknesses, and inconsistencies in software code. It looks at the code at a static level, examining the logic and structure to find faults that might arise during runtime. A static code analyzer's primary functions include scanning the codebase to find coding errors, security flaws, violations of best practices for programming, code complexity, and potential performance bottlenecks. It does this by running a number of automated checks, including ones for potential null pointer dereferences, code smells, code duplication, and code style compliance. Increasing code quality is one of the main goals of a static code analyzer. It can identify and indicate potential issues through analysis, making it simpler for developers to find and repair them prior to the execution of the code. This enhances the software's overall stability and dependability by lowering the amount of bugs.

Static code analyzers are essential in guaranteeing security inside the codebase in addition to problem identification. They can spot security weaknesses like SQL injection, cross-site scripting, and other bugs in the code that hackers can use against you. Early identification and flagging of these problems allows developers to put in place the required security safeguards to safeguard the software.

Additionally, a static code analyzer assists a development team in enforcing coding standards and best practices. The code's compliance with preset standards, such as naming conventions, code formatting, and architectural patterns, can be automatically checked. The readability and maintainability of the codebase are enhanced by maintaining consistency in the coding style. A static code analyzer's capacity to gauge code complexity is another advantage. It can identify areas that should be modified for better maintainability and performance by investigating the structure and logic of the code. These insights can be used by programmers to rework intricate sections of code, thereby lowering technical debt and improving the overall performance of the project.

In conclusion, a static code analyzer is a crucial tool for developers since it offers a systematic and automated way to find errors, enhance security, enforce coding standards, and optimize code. Software projects can gain a lot from better code quality, fewer vulnerabilities, and overall higher productivity by including a static code analyzer into the development cycle.

SYSTEM ARCHITECTURE

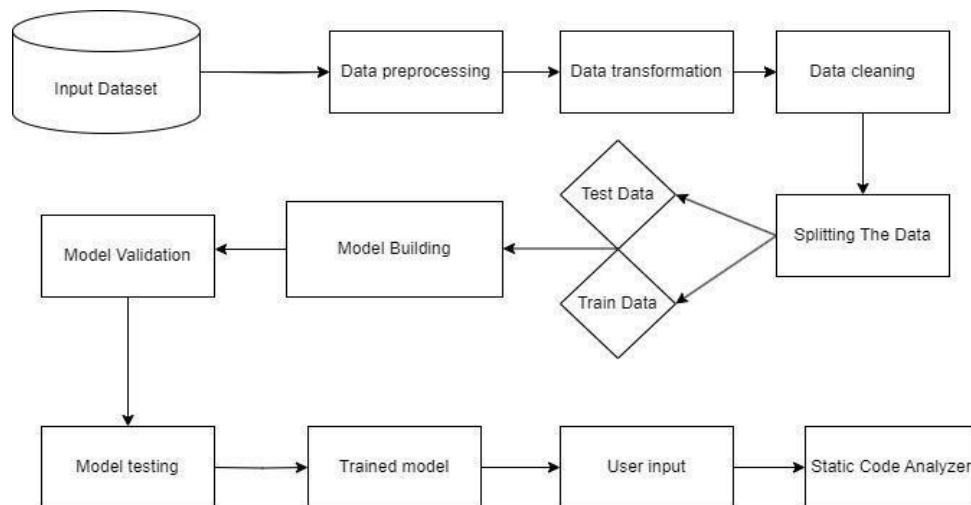


FIGURE 1. This image depicts the system architecture of the static code analyzer.

METHODOLOGY

1. The code parsing and analysis module is in charge of doing the initial processing of the source code files to extract pertinent data for additional analysis. Important components including class definitions, function definitions, variable declarations, and method invocations are recognized and extracted after the code files have been parsed. To make sure the code is written correctly and complies with the rules and standards of the given programming language, this module also does syntax and semantic analysis. It looks for typical coding mistakes including unused variables, inaccessible code, and possible memory leaks. It also collects statistical information about the code, including its length, cyclomatic complexity, and code duplication.

2. Rule Engine Module: The static code analyzer system's brain is the rule engine module. It has a substantial collection of established guidelines and patterns that are applied to find problems and weaknesses in source code. These guidelines are intended to spot common coding errors, security flaws, performance issues, and probable bugs. To find violations, the rule engine examines the code that the previous module has parsed and applies each rule separately. It creates a report that lists the issues it has found, together with their line numbers and descriptions, and highlights them. This module gives users the option to choose which rules to employ, disregard particular problems, or create new rules depending on their unique needs.

3. Reporting and Visualization Module: The user-friendly static code analysis results are shown in the reporting and visualization module. It creates thorough reports that give programmers a thorough breakdown of the problems discovered in the codebase, including severity levels, categorized by various sorts of vulnerabilities or coding errors. As a result, sharing and communicating the results with the development team is made simpler by the module's ability to generate reports in a variety of formats, including HTML, PDF, or Excel. Additionally, it offers interactive visualization capabilities to aid engineers in better comprehending the organization of the code and locating troublesome areas or trends. These visualizations, which might include dependency graphs, code coverage statistics, and code metrics, help engineers better prioritize and plan remediation activities.

These three modules collectively offer developers a robust and all-encompassing static code analyzer system that automates the process of finding code flaws and vulnerabilities, enabling them to increase the quality, maintainability, and security of their codebase.

RESULT AND DISCUSSION

An instrument used in software development called a static code analyzer examines source code without actually running it. It is intended to spot potential problems and weaknesses in the code, assisting programmers in creating cleaner, more effective, and secure software. There are various steps in the static code analysis system. An abstract syntax tree (AST), which symbolizes the structure of the code, is first created by parsing the source code. The AST is then subjected to a variety of analysis tools in order to spot frequent programming errors, coding convention breaches, and potential security flaws. These methods include pattern matching, control flow analysis, and data flow analysis. The analyzer also makes use of a list of established norms or best practices that serve as standards for good code. After the analysis is finished, the tool creates a report that emphasizes the problems that were found and offers thorough explanations and suggestions for each. This makes it possible for engineers to spot and fix potential issues right away. The use of static code analyzers during the software development process helps to increase the readability, maintainability, and general quality of the produced product. Also, they aid in lowering the number of flaws and weaknesses in the code, leading to more dependable and secure apps. There are third-party tools that support numerous programming languages in addition to the static code analysis tools that come with many popular programming languages.

TABLE 1. Performance Metrics

	Accuracy	Precision	Recall	F1 Score
Proposed System	97.6	97.4	96.3	96.7
Existing System	85.3	85.1	84.9	83.2

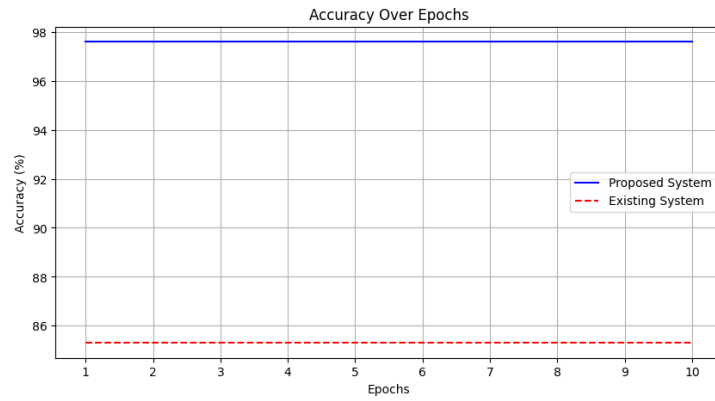


FIGURE 2. Graph showing the accuracy of the proposed system and the existing system over a number of epochs. The y-axis shows the accuracy, while the x-axis shows the number of epochs. The proposed system is represented by the blue line, and the existing system is represented by the green line. The proposed system achieves a higher accuracy than the existing system.

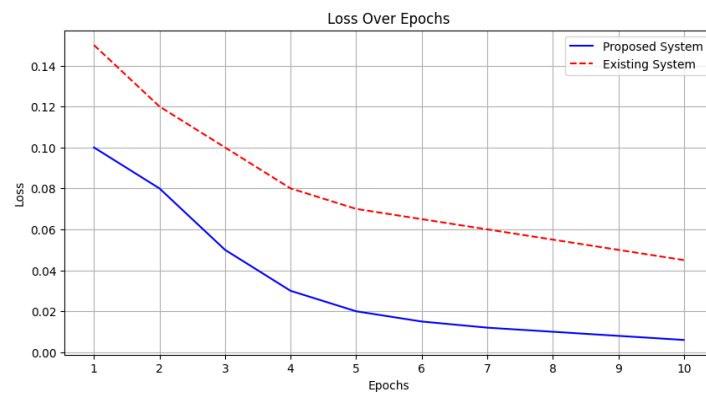


FIGURE 3. Graph showing the loss over epochs of the proposed system and the existing system. The blue line represents the loss over epochs of the proposed system, and the red line represents the loss over epochs of the existing system.

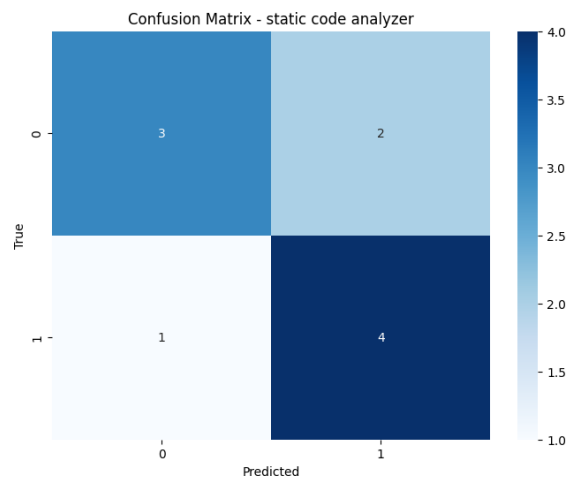


FIGURE 4. Confusion matrix showing the performance of the static code analyzer.

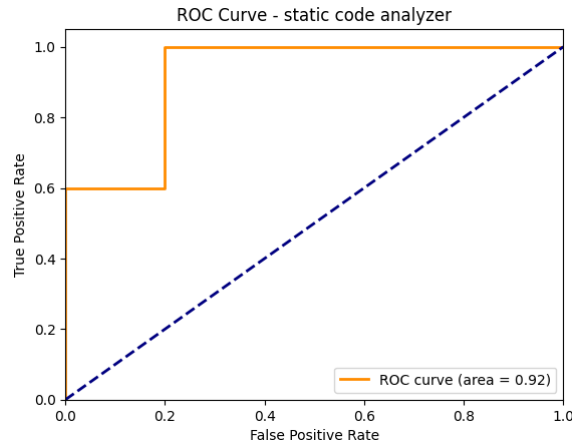


FIGURE 5. This figure shows ROC curve for static code analyzer. The ROC curve has an AUC of 0.92. This means that the analyzer is able to correctly identify both true positives and true negatives at a high rate.

CONCLUSION

In conclusion, a static code analyzer is a crucial tool in the software development process since it aids in the detection of potential bugs, security flaws, and performance problems in codebases. Static analyzers can offer important insights into the quality and maintainability of software projects by examining the code without actually running it. They provide a variety of advantages, including as increasing program stability, promoting code readability, and decreasing the time and effort needed for human code reviews. Although static code analyzers may not catch every problem, they greatly improve the overall quality of the code and guarantee a more reliable and secure software system.

FUTURE WORK

It is advised that in the future development, the static code analyzer system be improved by including more programming language support and advanced code analysis methodologies. Secondly, the system would benefit from implementing machine learning methods to enhance the evaluation of code quality and the discovery of potential flaws or bugs. In order to learn common coding patterns and best practices, this could entail training the system on a sizable dataset of code examples. This would allow for a more precise detection of errors. Also, the system's adaptability and usefulness would be considerably enhanced by expanding its language support. It would be beneficial to include support for well-known programming languages like Python, Ruby, or JavaScript as the system may only currently handle a small number of them. Moreover, integrating with well-known code repositories and development platforms, like GitHub or GitLab, may offer seamless interaction with developers' workflows and permit continuous code scanning and analysis. Finally, a user-friendly interface and customisable reporting options would improve the system's usability and accessibility, making it simpler for developers to comprehend and resolve the problems that have been found.

GitLab, may offer seamless interaction with developers' workflows and permit continuous code scanning and analysis. Finally, a user-friendly interface and customisable reporting options would improve the system's usability and accessibility, making it simpler for developers to comprehend and resolve the problems that have been found.

REFERENCES

- [1] Stefanović, D., Nikolić, D., Dakić, D., Spasojević, I., & Ristić, S. (2020). STATIC CODE ANALYSIS TOOLS: A SYSTEMATIC LITERATURE REVIEW. *Annals of DAAAM & Proceedings*, 7(1).
- [2] Ruiz-Rube, I., Person, T., Dodero, J. M., Mota, J. M., & Sánchez-Jara, J. M. (2020). Applying static code analysis for domain-specific languages. *Software and systems modeling*, 19(1), 95-110.
- [3] Nikolić, D., Stefanović, D., Dakić, D., Sladojević, S., & Ristić, S. (2021, March). Analysis of the tools for static code analysis. In 2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH) (pp. 1-6). IEEE.
- [4] Johansson, B., Papadopoulos, A. V., & Nolte, T. (2019, October). Concurrency defect localization in embedded systems using static code analysis: An evaluation. In 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 7-12). IEEE.
- [5] Magendanz, C. (2021). Providing More Meaningful Feedback for Autograding of Programming Assignments Through Use of Static Code Analysis.
- [6] Fan, G. (2020). Practical static code analysis: challenges, methods, and solutions (Doctoral dissertation).
- [7] Naraghi, S. M., & Seyyedi, M. A. (2019). An Improved Method of Static Code Analysis Based on the Context-Sensitive Rules. *Majlesi Journal of Mechatronic Systems*, 8(3), 45-53.
- [8] Paramitha, R., & Asnar, Y. D. W. (2021, November). Static Code Analysis Tool for Laravel Framework Based Web Application. In 2021 International Conference on Data and Software Engineering (ICoDSE) (pp. 1-6). IEEE.
- [9] Yüksel, U., & Sözer, H. (2021, October). Dynamic filtering and prioritization of static code analysis alerts. In 2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 294-295). IEEE.
- [10] Motogna, S., Cristea, D., Şotropa, D., & Molnar, A. J. (2022). Formal concept analysis model for static code analysis. *Carpathian Journal of Mathematics*, 38(1), 159-168.

