MANIPAL
ACADEMY of HIGHER EDUCATION
(Institution of Eminence Deemed to be University)

MANIPAL SCHOOL OF INFORMATION SCIENCES
(A Constituent unit of MAHE, Manipal)

Static Code Analyser Tool

Reg. Number	Name	Branch
241059027	Anusha S Patil	CS
241059032	K Anusha Rao	CS
241059033	S Srinivas	CS

Under the Guidance of

Mr. Satyanarayan Shenoy

Assistant Professor,

Manipal School of Information Sciences,

MAHE, MANIPAL

21/02/2025

MANIPAL SCHOOL OF INFORMATION SCIENCES
MANIPAL
(A constituent unit of MAHE, Manipal)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of software engineering is continually evolving, focusing on enhancing the quality, reliability, and maintainability of software [1]. Good code quality is a fundamental need in today's modern world [2]. Since source code is used universally to develop every software that runs on an electronic device, it is essential that source code must be well written and maintained in order to handle updates and enhancements.

Python is one of the most rapidly emerging programming language. Being flexible and expressive, it is very popular to implement Machine Learning and Cloud based systems among others [3]. The popularity is partially derived from its dynamic behavior: Python is a dynamically typed programming language, i.e. a variable is just a value bind to a certain (variable) name, the value has a type but not the variable. One can assign to an existing variable a new value with a possibly different type. This would be a compile time error in statically typed languages such as Pascal, C or Java, but allowed in Python. However, such dynamic behavior might be an obstacle when we try to validate the software systems written in Python.

When a software developer or code security analyst needs to analyze source code to detect security flaws and maintain secure, high-quality code, manual review may not uncover all issues. Even experienced security analysts can overlook vulnerabilities. A source code analysis tool addresses this challenge by automatically and efficiently identifying potential security flaws without executing the code. It serves as a reliable complement to manual inspection, enhancing accuracy and thoroughness. A Static Code Analyzer is a software tool that examines source code for potential errors, vulnerabilities, and adherence to coding standards without executing the program.

Chapter 2

Objectives

- Identify common code defects and vulnerabilities
- Develop a basic static code analysis tool
- Design an interface for users to upload, scan and review code analysis reports

Chapter 3

Literature Survey

The literature survey gives a brief overview about the various security vulnerabilities which can be detected by a static code analyzer and also about the existing tools available. The primary focus is to gather information on how these tools contribute to improving code security by identifying potential flaws such as cross-site scripting (XSS), SQL injection, broken authentication, and other vulnerabilities listed in the OWASP Top 10.

In [1], Hassan *et al.* has evaluated the Python static code analysis tools using FAIR principles to improve usability. The research highlights the strengths and limitations of each tool in terms of findability, accessibility, interoperability, and reusability. This study holds practical significance as it bridges theoretical evaluation frameworks with actionable insights for developers, researchers, and organizations. By connecting Python static code analysis tools with FAIR principles, this research helps software developers make smarter choices when choosing the right tools. As a future development, a web application is proposed to help developers choose and compare tools based on their programming language, analysis needs, and ease of integration.

In [2], Peiris *et al.* develops a tool designed to enhance developer productivity by detecting structural code issues through static analysis. The objectives are to improve coding practices by identifying potential problems early in the development process. The study suggests future improvements, including the refinement of the algorithm to detect complex issues such as tightly coupled code and collusion. Furthermore, integrating AI and ML techniques could enhance issue detection and predictive capabilities, making the tool more efficient and valuable to developers.

In [3], Gulabovska *et al.* proposed symbolic execution as an alternative to traditional static analysis methods, demonstrating its ability to improve the accuracy of error detection. Using

symbolic execution, the research highlights how this technique can analyze code paths more effectively than conventional static analysis methods. The study suggests that future tools could incorporate powerful SAT solvers, such as Z3, to further improve precision, automate complex code evaluations, and provide better vulnerability detection for Python programs.

In [4], Dong & Chen *et al.* proposed PType, a static type analysis tool designed to improve Python's type checking by automatically annotating classes, functions, and built-in modules. The tool constructs a constraint graph to infer types and detect potential inconsistencies in code-bases. The study emphasizes the importance of type inference in large-scale Python projects where dynamic typing can lead to unexpected runtime errors. Future enhancements include expanding PType's type-checking capabilities and integrating it with other static analysis tools to provide a more comprehensive solution for developers working on complex Python applications.

In [5], Ruohonen *et al.*focus on analysing Python packages from the Python Package Index (PyPI) to identify security vulnerabilities using static analysis techniques. By employing Bandit, a security-focused static analysis tool, the research uncovers common security risks in widely used Python packages. The findings suggest that many packages suffer from weak security practices, making them susceptible to attacks. They recommend collecting software metrics to refine vulnerability prediction models and suggest longitudinal analysis to monitor how security risks evolve over time. These insights can help improve security policies and best practices for open-source Python package development.

Chapter 4

Specifications

4.1 Introduction

4.1.1 Purpose

The purpose of this Software Requirements Specification (SRS) document is to define the functionalities, performance, and design requirements of a static code analyzer. This tool aims to assist developers, security analysts, and organizations in identifying and reporting security vulnerabilities in the code base.

4.1.2 Scope

The static code analyzer helps ensure code quality, security, and compliance by detecting issues such as syntax errors, unused variables and insecure API calls. It plays a crucial role in identifying vulnerabilities like SQLi, hardcoded credentials, and improper input validation early in development. By integrating static analysis into the development workflow, teams can enhance code reliability, reduce debugging effort, and meet security standards like OWASP guidelines.

4.1.3 Abbreviations

The Abbreviations of the words used in this report are shown in the Table 4.1.

OWASP	Open Web Application Security Project
SQLi	SQL Injection
AST	Abstract Syntax Tree

Table 4.1: Abbreviations

4.2 Functional Requirements

- **Code Defect Detection:** Identifies syntax errors, unused variables and improper indentation.
- **Security Vulnerability Detection:** Scans for SQL injection, hardcoded secrets, and unsafe function calls.
- **Automated Code Scanning:** Parses and analyzes Python scripts without execution.
- **Report Generation:** Provides structured reports in JSON, HTML, or CSV formats.
- **User Interface for Code Review:** Enables users to upload, scan, and view reports in a web-based interface.
- **Error Classification:** Categorizes issues into warnings, errors, and security threats.

4.3 Non-Functional Requirements

- **Usability:** Provides a simple and interactive UI for code scanning and report review.
- **Security:** Ensures safe handling of uploaded code and prevents unauthorized access.
- **Compatibility:** Supports Python 3.x and integrates with IDEs like VS Code and PyCharm.
- **Error handling:** The system should handle failed scans gracefully and provide detailed error logs.

Chapter 5

Proposed Methodology

The methodology for the Static Code Analyzer is based on the structured process illustrated in figure 5.1. It follows a sequential approach for analyzing Python code, identifying vulnerabilities, and generating reports.

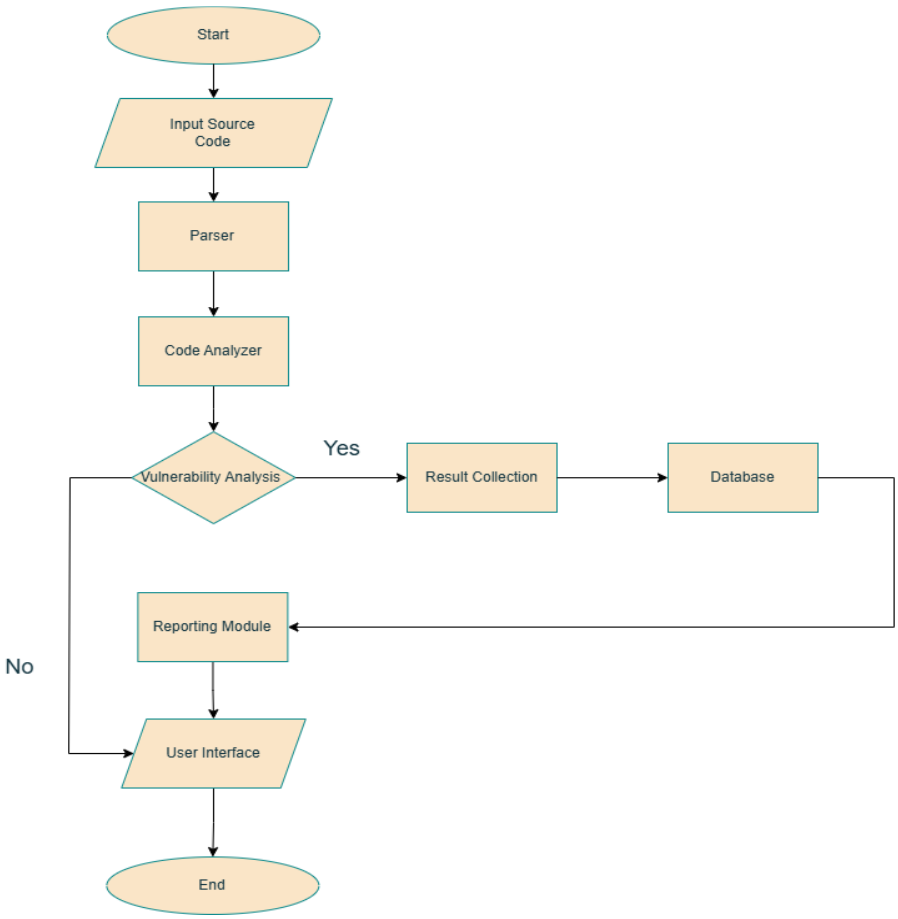

Figure 5.1: Flowchart of static code analyzer

- **Input Source Code**

  - Users upload Python source code via the tool's interface.
  - The system accepts files in .py format for analysis.

- **Parsing**
  - The uploaded code is tokenized and parsed using Python's built-in AST module.
  - The parser converts the code into a structured format for further analysis.

- **Code Analysis**
  - The Static Code Analyzer inspects the parsed code for:
    - **Syntax errors**
    - **Security vulnerabilities**
    - **Unused variables**

- **Vulnerability Analysis**
  - A decision point checks whether any vulnerabilities or defects are detected.
  - **If vulnerabilities exist:**
    - The results are collected and stored in a database.
  - **If no issues are found:**
    - The system proceeds directly to report generation.

- **Result Collection & Database Storage**
  - Detected vulnerabilities are categorized (e.g., critical, high, medium, low).
  - The results are logged in the database for future reference and auditing.

- **Reporting Module**
  - Generates a structured report with details of Defects found and Severity levels.
  - The report format may include JSON, HTML, or CSV.

- **User Interface & Report Review**
  - Users access the web-based interface to review reports.
  - The report format may include JSON, HTML, or CSV.

Chapter 6

Work Done

Project planning is a procedural step in the project management, where required documentation is created to ensure successful project completion. A work plan represents the formal road map for a project. The project planning for proposed system is shown in the Figure 6.1

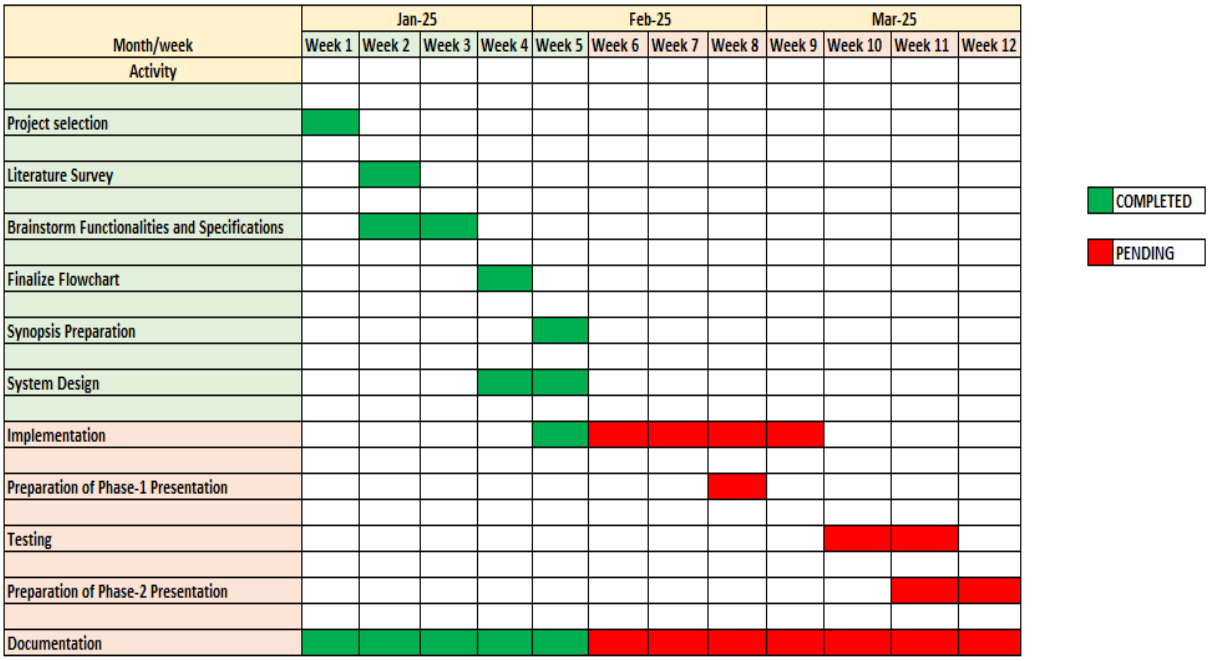

Figure 6.1: Gantt Chart of Project Timeline

6.1 Week 1

6.1.1 Requirement Finalization and Project Setup

During the first week, we finalized the core requirements and features of the static code analyzer. This involved defining the functionalities, scope, and expected outcomes of the project.

A structured folder hierarchy was established for better organization of the frontend and back-end components. Additionally, we set up a GitHub repository with an initial README file to document the project tasks. This foundational work ensured that all team members had a clear understanding of the project structure and workflow.

6.2 Week 2

6.2.1 Technology Stack Setup and Literature Survey

In the second week, we initialized the frontend using React and the backend using FastAPI, running on Uvicorn. FastAPI was chosen for its high performance, asynchronous support, and seamless integration with modern web technologies. The frontend was set up to communicate with the backend via API requests. Alongside development, we conducted a literature survey, researching existing static code analysis tools and their methodologies. This research helped refine our approach and ensured our tool aligns with best practices. A project synopsis was also prepared, outlining objectives and technical decisions.

6.3 Week 3

6.3.1 Frontend-Backend Integration and File Upload Implementation

By the third week, we focused on building the file upload feature. The frontend was enhanced with a file selection and upload interface, allowing users to submit Python scripts for analysis. On the backend, we developed the /upload endpoint in FastAPI to receive and store uploaded files for processing represented in Figure 6.2 and Figure 6.3.

However, we encountered some integration challenges, particularly CORS issues while making API requests from the frontend to the backend. Troubleshooting these issues took additional time, but we successfully resolved them by configuring CORS middleware in FastAPI. While the basic UI for file uploads was functional, we still needed to refine the design and ensure error handling was properly implemented.

```python
def save_uploaded_file(file: UploadFile) -> str:
    """
    Save the uploaded file with a unique filename to avoid overwriting existing files.

    Author: K Anusha Rao
    Description: This function takes an uploaded file, generates a unique filename by
                 appending the current timestamp to the original filename, and stores it
                 in the specified directory. The function returns the file path of the
                 saved file.

    Args:
    - file (UploadFile): The uploaded file to be saved.

    Returns:
    - str: The file path where the file was saved.
    """

    # Generate a unique filename by appending timestamp to the original filename
    timestamp = int(time.time() * 1000)  # Milliseconds timestamp
    file_name = f"{timestamp}_{file.filename}" # Combining timestamp and original filename
    file_path = os.path.join(UPLOAD_DIR, file_name)

    # Write the file content to the specified file path
    with open(file_path, "wb") as buffer:
        buffer.write(file.file.read()) # Save the file

    return file_path
```

Figure 6.2: Save Uploaded File Logic

```python
@app.post("/upload/")
async def upload_file(file: UploadFile = File(...)):
    """
    Endpoint to handle file uploads.

    Args:
        file (UploadFile): The uploaded file from the client.

    Returns:
        dict: A message indicating success and the saved file path.
    """
    file_path = save_uploaded_file(file) # Save file with a unique name
    return {"message": "File uploaded successfully", "file_path": file_path}
```

Figure 6.3: Upload Endpoint Logic

6.4 Week 4 & 5

6.4.1 Stabilization, Mid-Presentation, and Enhancements

During the fourth and fifth weeks, we focused on stabilizing the file upload feature and improving the interaction between the frontend and backend. We conducted initial tests to verify that the file upload mechanism worked as expected. At this stage, we also prepared for the mid-project presentation, which included a summary of our progress, an overview of challenges encountered, and our plan for the remaining tasks.

One of the key tasks planned for Week 4 was to start implementing the static analysis module for detecting vulnerabilities and code defects. However, due to time constraints and debugging efforts in the previous weeks, we were unable to begin this phase. Additionally, we had planned to introduce basic report generation to display the analysis results, but this remains a pending task.

Chapter 7

Results

After five weeks of development, we have successfully built a functional file upload system that allows users to submit Python scripts for analysis represented in Figure 7.1. The frontend and backend integration is stable, with FastAPI (running on Uvicorn) handling requests efficiently. Despite delays, we have established a solid foundation for the static analysis engine, which will be implemented in the next phase. Initial testing confirms that the system correctly handles file uploads and stores them for processing. However, vulnerability detection, code defect identification, and report generation remain incomplete. The completed components work as expected, and we are well-positioned to implement the core analysis functionalities in the upcoming weeks.

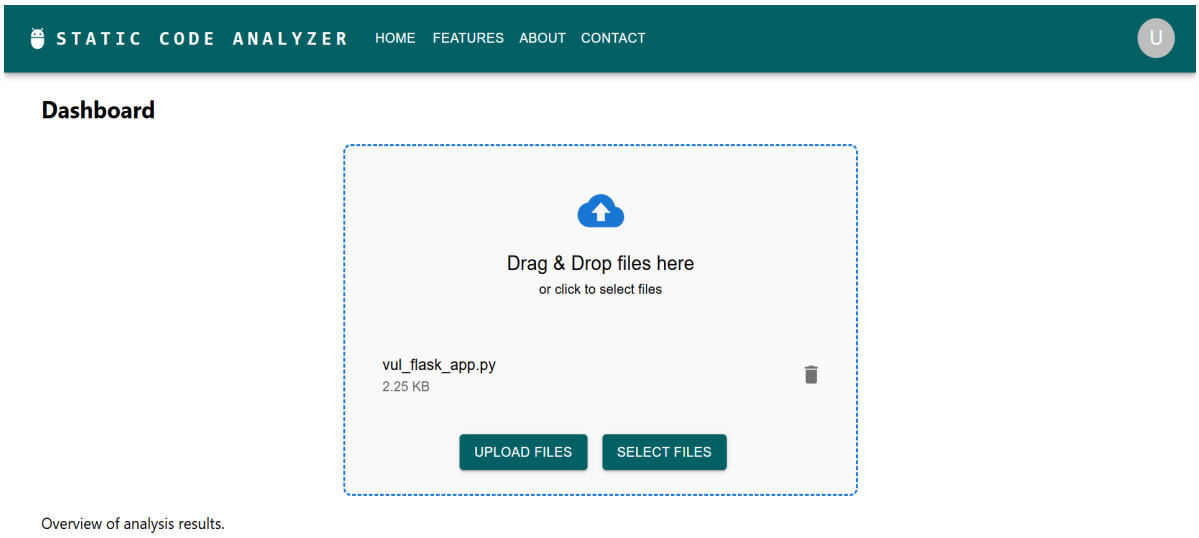

Figure 7.1: Basic UI for Static code Analyzer

Chapter 8

Conclusion

We have successfully established the backend infrastructure, frontend integration, and file upload mechanism, ensuring a stable foundation for further enhancements. The key functionalities of static code analysis, vulnerability detection, and report generation are yet to be developed, but with the groundwork in place, we are well-prepared to move forward. Our next steps will focus on implementing the analysis engine, refining the UI, and improving reporting features to ensure a fully functional and user-friendly static code analyzer.

References

[1] H. B. Hassan, Q. I. Sarhan and Á. Beszédes, ”Evaluating Python Static Code Analysis Tools Using FAIR Principles,” in IEEE Access, vol. 12, pp. 173647-173659, 2024.

[2] D. R. I. Peiris and N. Kodagoda, ”Static Code Analyser to Enhance Developer Productivity,” 2023 IEEE 8th International Conference for Convergence in Technology (I2CT), Lonavla, India, 2023, pp. 1-6.

[3] H. Gulabovska and Z. Porkoláb, ”Towards More Sophisticated Static Analysis Methods of Python Programs,” 2019 IEEE 15th International Scientific Conference on Informatics, Poprad, Slovakia, 2019, pp. 000225-000230.

[4] T. Dong, L. Chen, Z. Xu and B. Yu, ”Static Type Analysis for Python,” 2014 11th Web Information System and Application Conference, Tianjin, China, 2014, pp. 65-68.

[5] J. Ruohonen, K. Hjerppe and K. Rindell, ”A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI,” in 2021 18th International Conference on Privacy, Security and Trust (PST), Auckland, New Zealand, 2021, pp. 1-10.