**Bug Detection and Fixing**

A Project Report

submitted in partial fulfillment of the requirements

of

Intel® Unnati Industrial Training 2025

by

**Aishika Majumdar** ([bwubta22403@brainwareuniversity.ac.in](mailto:bwubta22403@brainwareuniversity.ac.in))

**Saikat Patra** ([bwubta22412@brainwareuniversity.ac.in](mailto:bwubta22412@brainwareuniversity.ac.in))

**Pritam Chakrabortty** ([bwubta22388@brainwareuniversity.ac.in](mailto:bwubta22388@brainwareuniversity.ac.in))

**Soumavo Acharjee** ([bwubta22401@brainwareuniversity.ac.in](mailto:bwubta22401@brainwareuniversity.ac.in))

Under the guidance of

**Dr. Shivnath Ghosh, HOD, CSE-AI, Brainware University**

**Acknowledgement**

**Abstract**

Software bugs pose significant challenges in software development, leading to inefficiencies, security vulnerabilities, and increased costs. This project presents an AI-powered Bug Detection and Fix Recommendation System that automates the identification and resolution of code errors using deep learning and natural language processing (NLP) techniques. The system leverages CodeBERT, a pre-trained transformer model, to detect bugs in Python code and classify them into various categories such as syntax errors, logic errors, and runtime issues.

Once a bug is identified, the system integrates with Google Gemini AI to provide intelligent fix recommendations, ensuring that developers receive precise and context-aware solutions. The project consists of data preparation, model training, bug classification, and automated bug resolution. Our approach improves debugging efficiency by reducing manual efforts and enhancing code reliability. The system is evaluated using standard metrics such as accuracy, precision, recall, and F1-score, demonstrating its effectiveness in real-world scenarios.

This project contributes to the growing field of AI-assisted software engineering by automating bug detection and providing actionable insights, making software development more efficient and error-free.

**CONTENTS**

**Abstract I**

**Chapter 1.**  **Introduction 1-4**

1.1 Problem Statement 1

1.2 Motivation 1

1.3 Objectives 2

1.4. Scope of the Project 3

**Chapter 2.**  **Literature Survey 5-7**

2.1 Review of Relevant Literature .…………………………………...….5

2.2 Existing Models, Techniques and Methodologies…...…………...…..5

2.3 Gaps and Limitations in existing solutions…………………………...6

**Chapter 3.**  **Proposed Methodology 8-10**

3.1 System design…………………………………………………………8

3.2 Requirement Specification ………….………………………………..9

**Chapter 4.**  **Implementation and Results 11-12**

4.1 Snapshot of result…………………………………………………….11

4.2 Github link for code………………………………………………….12

**Chapter 5. Discussion and Conclusion 13-14**

5.1 Future scope………………………………………………………….13

5.2 Conclusion…………………………………………………………...14

**References** 15

1. **Introduction**
   1. **Problem Statement**

Software bugs can lead to security vulnerabilities, system failures, and increased maintenance costs. Traditional debugging methods are time-consuming and often rely on manual intervention. This project aims to develop an AI-powered Bug Detection and Fix Recommendation System that can automatically identify and rectify errors in source code.

The system will leverage machine learning and transformer-based models to detect bugs across multiple programming languages, classify their types, and generate intelligent fix recommendations. By combining static code analysis, deep learning, and natural language processing (NLP), this solution will enhance developer productivity and streamline the debugging process. The model's effectiveness will be evaluated using standard metrics such as precision, recall, and F1-score, ensuring accuracy and reliability in real-world software development.

This project contributes to AI-assisted software engineering, reducing debugging time and improving code quality through an automated and intelligent bug-fixing approach.

* 1. **Motivation**

Software development is prone to errors, with bugs ranging from minor syntax mistakes to critical logic flaws that can cause system crashes and security vulnerabilities. Debugging is often a time-consuming, manual process that requires significant effort from developers. Traditional methods rely on static code analysis tools, which may not always catch deeper logical or semantic errors.

With the rise of AI and machine learning, automated bug detection and fix recommendation systems offer a promising solution. This project is motivated by the need for a faster, intelligent, and automated debugging process that enhances code quality and reduces development time. By leveraging deep learning and NLP-based models, we aim to create a system that can identify, classify, and correct code errors, making software development more efficient and reliable.

* 1. **Objectives**

The key objectives of this project are:

**1. Data Collection & Preparation**

- Gather code snippets one programming language.

- Label them as “buggy” or “bug-free” and include corrected versions for training.

**2. Bug Detection Model**

- Develop a machine learning model that can analyze code structure and detect potential errors.

- Train the model using transformer-based architectures (CodeBERT).

**3. Fix Recommendation System**

- Implement an AI-powered suggestion mechanism that proposes fixes for detected bugs.

- Ensure that recommendations are context-aware and improve code correctness.

**4. Evaluation & Performance Metrics**

- Use precision, recall, F1-score to assess the model’s bug detection capability.

- Evaluate the accuracy and effectiveness of fix recommendations.

**5. Usability**

- Use an API for developers to input code and receive automated bug detection and fixes.

- Ensure the system generalizes across different coding styles and libraries.

* 1. **Scope of the Project**

The Bug Detection and Fix Recommendation System has a broad scope in modern software development, covering:

* **Automated Debugging**: Reducing manual debugging time by providing quick and accurate error detection.
* **Python Language Support**: Initially focusing Python language with potential expansion.
* **AI-Powered Code Assistance**: Helping both novice and experienced developers improve their code quality.
* **Integration with Development Tools**: The system can be extended to work as a plugin for IDEs (e.g., VS Code, PyCharm) or integrated into CI/CD pipelines in the future.

**Real-world Applications:** Useful for software companies, coding boot camps, and individual developers to maintain efficient and error-free codebases.

1. **Literature Survey**
   1. **Review of relevant literature**

*Traditional Bug Detection Approaches* ***-*** Bug detection has traditionally relied on static and dynamic code analysis techniques.

* Static Analysis Tools like **Lint, SonarQube,** and **FindBugs** analyze source code without execution, detecting syntactical errors, security vulnerabilities, and code smells.
* Dynamic Analysis involves running the program to detect runtime errors. Tools like **Valgrind** and **AddressSanitizer** help in memory leak detection and undefined behavior analysis.

However, these rule-based techniques often struggles with logical bugs and require manual effort to interpret results.

*Machine Learning for Bug Detection -* With the rise of AI-driven software engineering, ML models have been explored for bug detection:

* **DeepCode** (*Vassallo et al., 2020*): Uses deep learning to analyze code patterns and detect potential issues.
* **CodeBERT** (*Feng et al., 2020*): A transformer-based model trained on large-scale code datasets, effective in code completion and bug detection.
* **Graph Neural Networks (GNNs) for Code Analysis** (*Allamanis et al., 2018*): Leverages ASTs (Abstract Syntax Trees) to understand code structure and identify bugs.

*Fix Recommendation Systems -* Fix recommendation is an emerging area, often using seq2seq models and LLMs:

* **Tufano et al. (2019):** Proposed a sequence-to-sequence deep learning model for automated code fixes, using historical bug-fix pairs.
* **Codex (OpenAI, 2021):** A fine-tuned GPT model capable of bug fixing and code generation, influencing AI-powered coding assistants like GitHub Copilot.
* **Fault Localization + Repair (Gupta et al., 2017):** Combines fault localization techniques with ML-based fix generation, improving the accuracy of suggestions.
  1. **Gaps and Limitations in Existing Solutions**

While machine learning (ML)-based bug detection and fix recommendation systems have significantly advanced, they still face several challenges. Some of the key gaps and limitations are listed in the table below:

|  |  |  |
| --- | --- | --- |
| **Data Quality and Availability** | Lack of High-Quality Labeled Datasets | * Most available datasets contain noisy, incomplete, or incorrect labels, leading to poor model generalization. * Bug-fix pairs from open-source repositories are often unstructured, making supervised learning challenging. |
| Language and Framework Dependence | * Many existing solutions are language-specific, meaning a model trained on Python may not perform well on Java or C++. * Fix recommendation models often struggle with framework-specific bugs that require contextual understanding. |
| **Model Performance Limitations** | Poor Generalization to Unseen Code | * ML models often memorize patterns rather than truly understanding program logic. * Code structure varies widely between developers and projects, causing inconsistent predictions for real-world applications. |
| Challenges in Understanding Logical Errors | * Syntax errors are easy to detect, but identifying semantic and logical bugs (e.g., infinite loops, incorrect algorithms) is still an open challenge. * Current AI models lack program execution simulation, leading to inaccurate bug detection. |
| Fix Recommendations May Lack Context Awareness | * Suggested fixes are often syntactically correct but logically incorrect, as models focus more on surface-level patterns than deep program logic. * ML-based fixers struggle to differentiate between multiple possible correct fixes for the same bug. |
| **Computational Challenges** | High Computational Cost | * Transformer-based models like GPT require significant computational resources for training and inference. * Running these models in real-time environments, especially for large-scale projects, remains difficult. |
|  | Inference Time and Latency Issues | * Real-time bug detection and auto-fixing require fast inference speeds, which is a challenge for large models. * The trade-off between model complexity and efficiency affects practical deployment. |
| **Human** **Intervention Still Required** | Reliance on Developers for Verification | * While AI can detect and suggest fixes, human review is still necessary to ensure correctness and security. * Many fixes require manual adjustments, making full automation difficult. |
|  | Lack of Explainability | * Most ML models function as black boxes, meaning developers don’t understand why a particular bug was flagged or why a fix was suggested. * This reduces trust and limits widespread adoption in production systems. |
| **Security and Ethical Concerns** | Vulnerability to Adversarial Attacks | * ML models can be tricked by adversarial code samples, leading to false positives/negatives in bug detection. * If attackers understand how the model works, they can intentionally bypass detection. |
|  | Bias in Training Data | * Models trained on biased datasets may fail to detect certain types of bugs or suggest inappropriate fixes. * If not properly validated, ML-based fixers could introduce new security vulnerabilities instead of fixing them. |

* 1. **How our Project addresses these gaps**

Our project aims to build upon these advancements by:

* Leveraging transformer-based models ( CodeBERT ) for improved bug detection and Gemini API for fix recommendations.
* Addressing generalization issues by using diverse training data from multiple repositories.
* Enhancing fix accuracy with a hybrid approach combining ML-based prediction with static analysis validation.
* Providing justifications for bug detections and fix suggestions, increasing transparency.

1. **Proposed methodology**

The Bug Detection and Fix Recommendation System is designed to identify potential bugs in code and suggest appropriate fixes using Machine Learning (ML) and Deep Learning (DL) models. The proposed methodology involves multiple stages, including data collection, preprocessing, model training, evaluation, and deployment.

**Step 1: Data Collection and Preprocessing**

* Collect buggy and bug-free code snippets from various open-source repositories, coding forums, and datasets.
* Label code snippets as "buggy" or "bug-free" and, for buggy code, provide a corresponding fixed version.
* Use Abstract Syntax Trees (ASTs) and tokenization for structured code representation.
* Perform data augmentation techniques (e.g., introducing minor syntax errors) to enhance model robustness.

**Step 2: Feature Extraction and Representation -** Convert code into numerical representations using:

* Token-based encoding (Byte Pair Encoding).
* Graph-based representations (AST embeddings).
* Transformer-based models like CodeBERT.

**Step 3: Model Development**

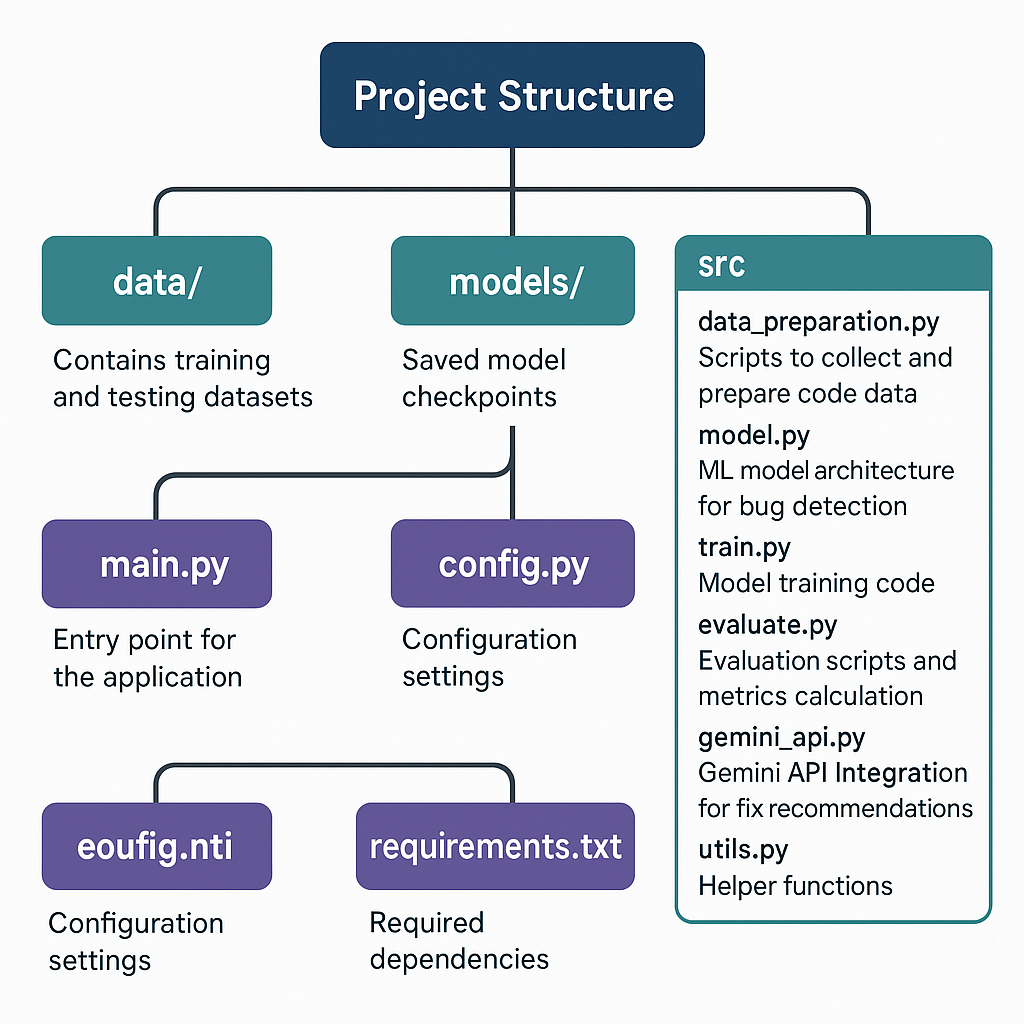
* Train a bug classification model using Deep Learning models (LSTMs, Transformers, BERT-like architectures).
* Use supervised learning for bug detection and sequence-to-sequence models (like T5 or GPT) for fix recommendations.
* Fine-tune existing models or train from scratch based on dataset availability.

**Step 4: Model Evaluation and Optimization -** Evaluate model performance using:

* Bug Detection Metrics: Precision, Recall, F1-score.
* Fix Recommendation Metrics: BLEU score, Edit distance.
* Optimize using hyperparameter tuning, dropout layers, and fine-tuning with reinforcement learning.

**Step 5: Utility**

* Convert the trained model into an API-based system for easy integration into development environments (e.g., VS Code extension, GitHub Actions).
* Implement user-friendly feedback mechanisms where developers can review, modify, and accept/reject suggested fixes.
  1. **System Design**

****

* 1. **Requirement Specification**

**Software Requirements**

1. Programming Language: Python
2. Frameworks & Libraries:

* PyTorch / TensorFlow for deep learning
* Scikit-learn for traditional ML
* NLTK / Hugging Face Transformers for NLP models

1. Version Control: GitHub for collaborative development
2. Data Storage: Local Storage of at least 100 GB

**Hardware Requirements**

1. Processor: Minimum Intel Core i7 / AMD Ryzen 7
2. RAM: At least 16GB (for training large models)
3. GPU: NVIDIA RTX 3060 or higher (for deep learning acceleration)
4. Storage: SSD with at least 100GB of free space

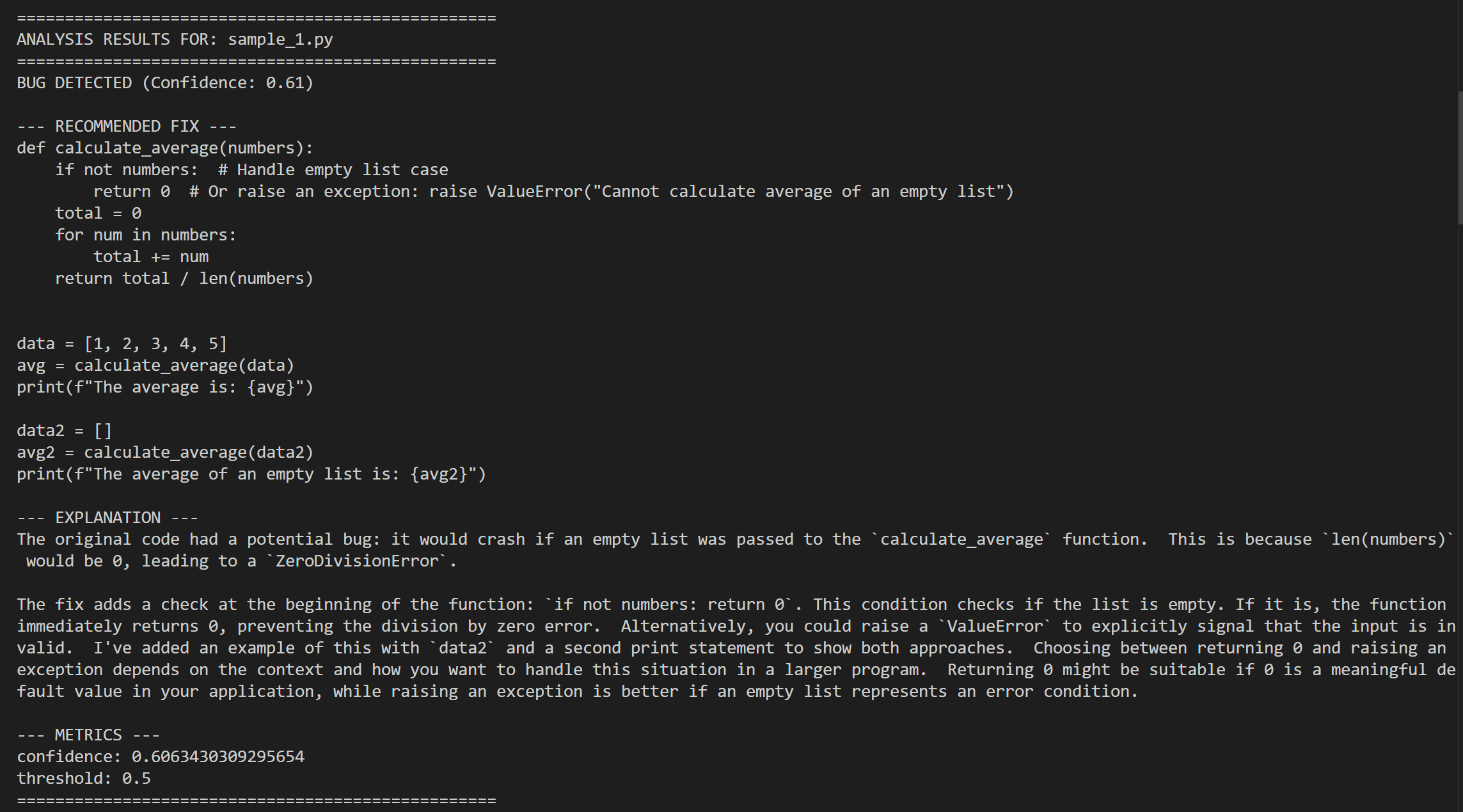
**Security Measures:**

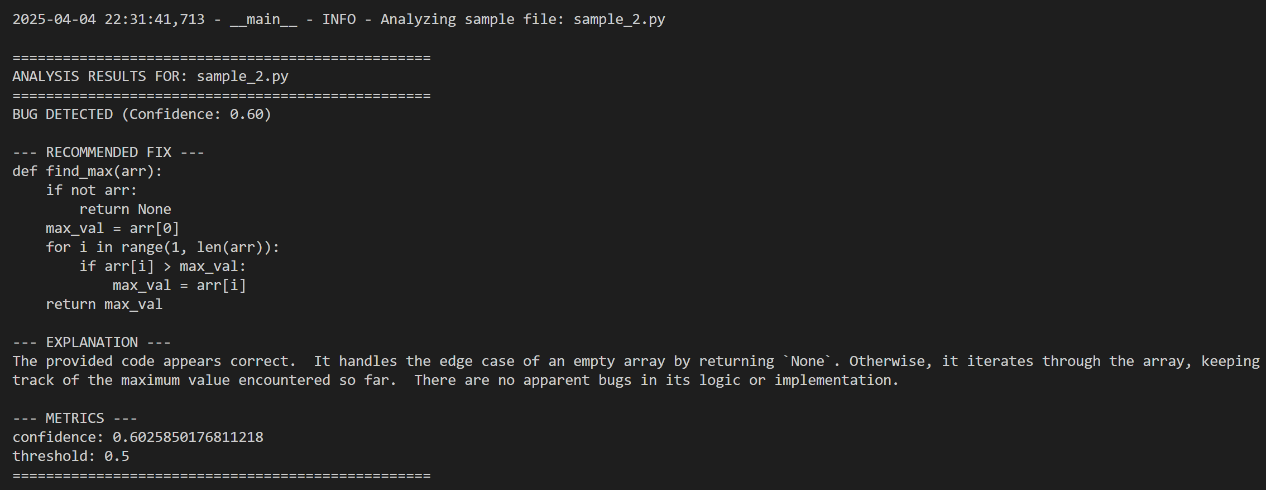
* Authentication & Authorization for API access
* Input validation to prevent code injection attacks

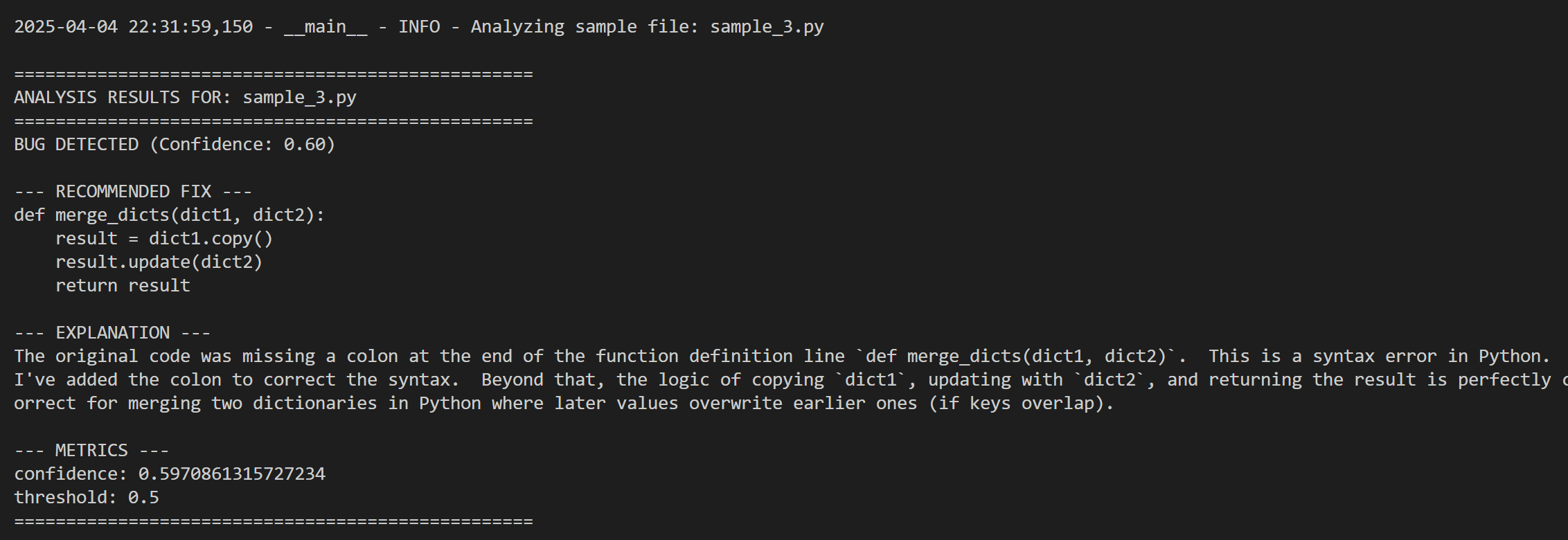
**4. Implementation and Results**

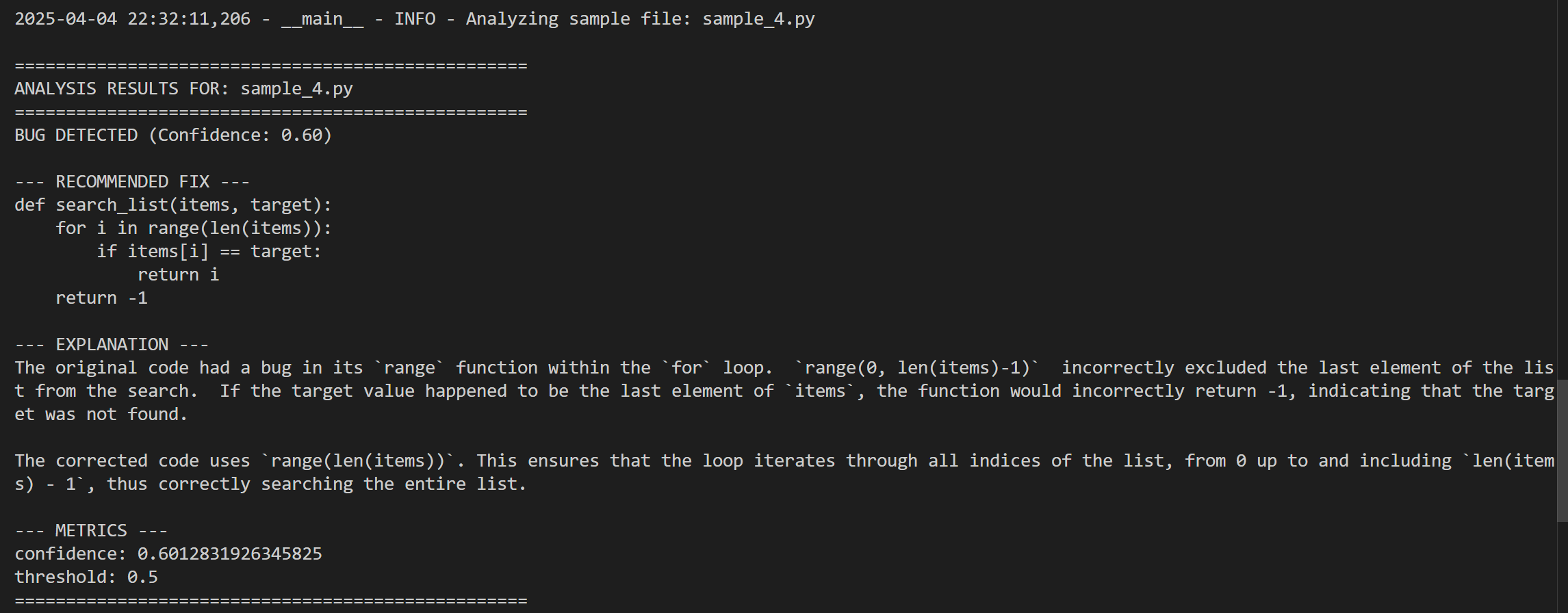
**4.1 Snapshot of Result**

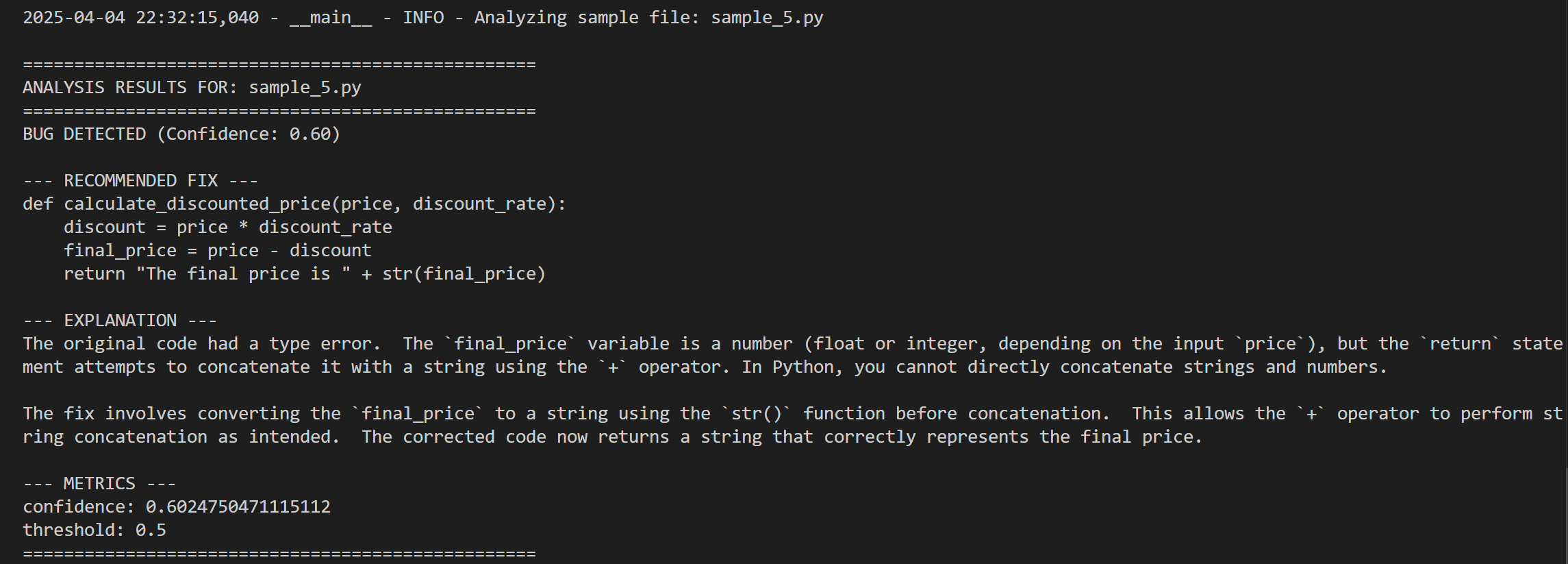
**🡪** On running this command – ‘python main.py’ the all code samples inserted in the ‘samples’ folder will automatically start to be analyzed by the model. The followings are the snapshots the output after analysis of each code sample at a time --

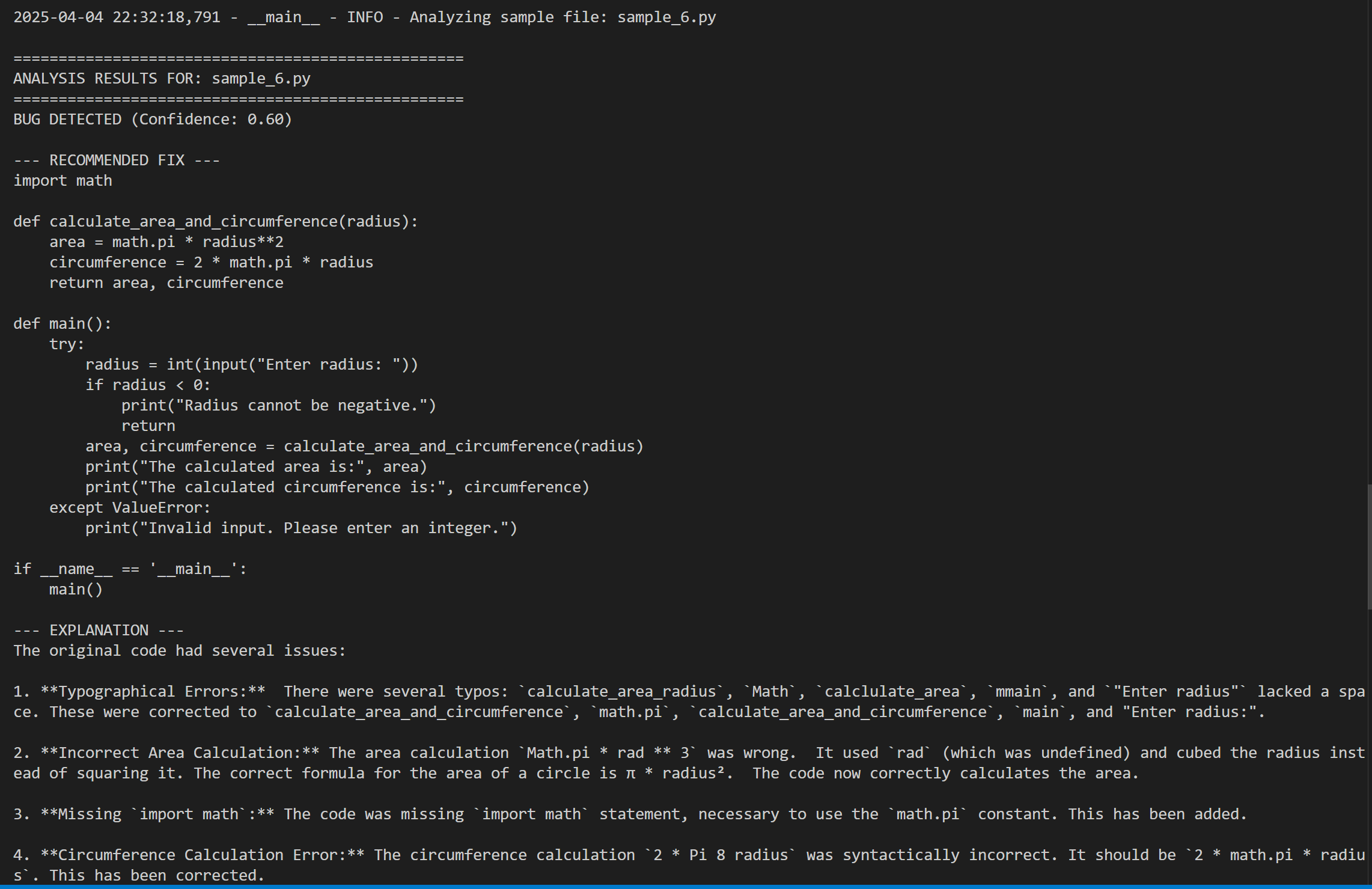
****

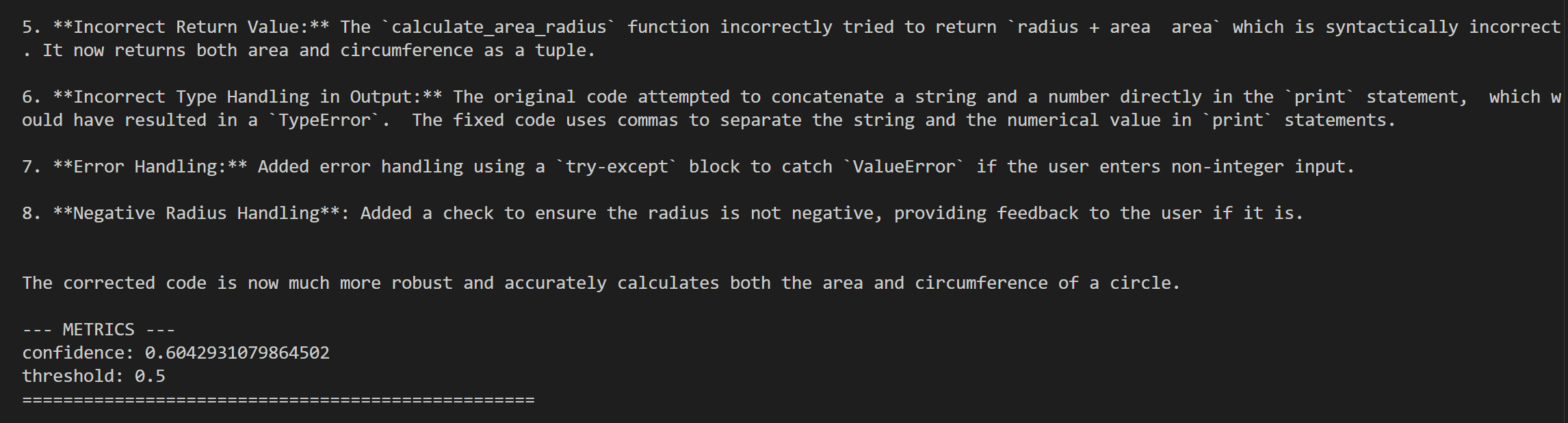
****

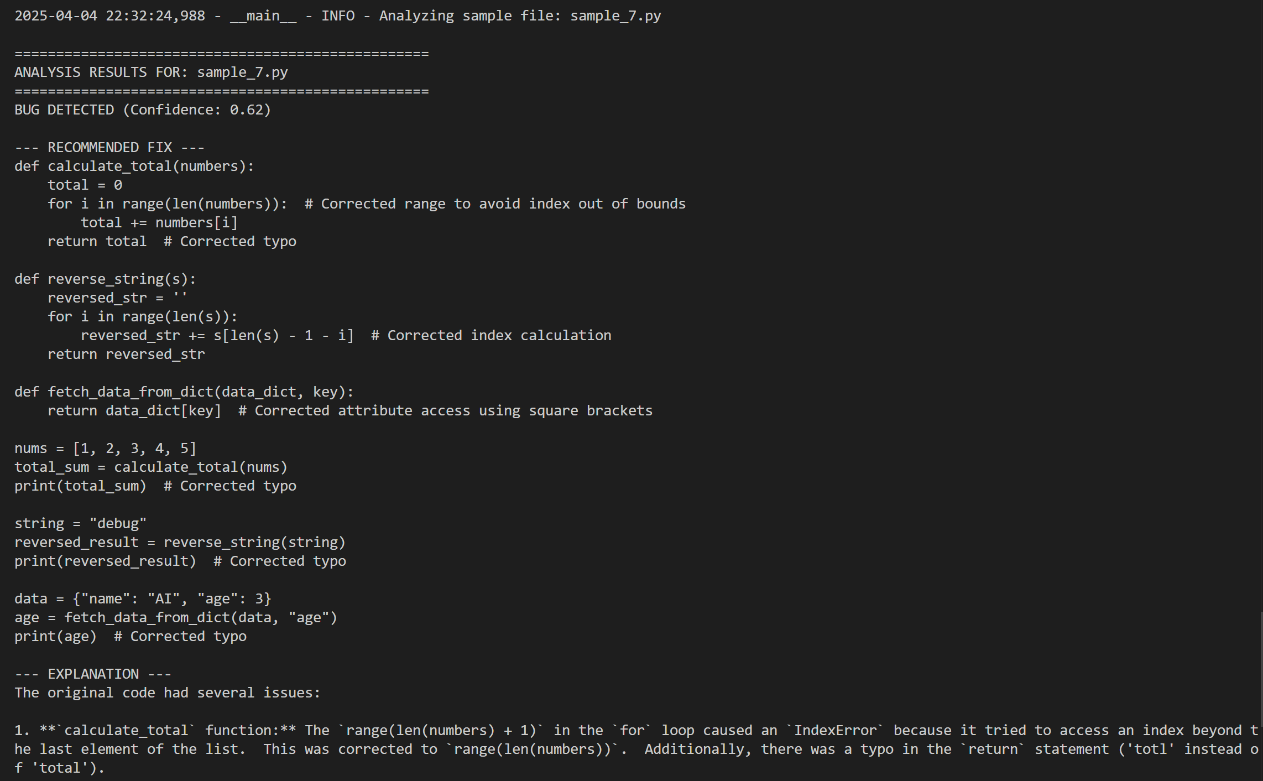
****

****

****

****

****

****

* 1. **GitHub link for Code**

1. **Discussion and Conclusion**

**5.1 Future Scope**

The Bug Detection and Fix Recommendation System has the potential for significant improvements and expansions in the future. Some key areas of growth include:

1. **Support for More Programming Languages**

* Currently, the system may be optimized for a specific language (Python). Future iterations can extend support to C, C++, Java, JavaScript, Go, Rust, and other languages to enhance its versatility.

1. **Integration with Developer Tools & IDEs**

* The model can be embedded into popular development environments such as Visual Studio Code, PyCharm, IntelliJ, and JetBrains, enabling real-time bug detection and fix suggestions during coding.

1. **AI-Powered Code Auto-Fix**

* Implement self-correcting code that not only detects errors but also applies fixes autonomously, reducing manual debugging efforts.

1. **Context-Aware Fix Recommendations**

* Future improvements could incorporate knowledge of code repositories, project structures, and software engineering best practices to provide more accurate fixes based on the specific application context.

1. **Integration with CI/CD Pipelines**

* The system can be integrated into Continuous Integration/Continuous Deployment (CI/CD) workflows, ensuring automated bug detection before deployment. This will help software teams maintain high code quality with minimal human intervention.

1. **Explainable AI for Debugging**

* Enhancing the system with Explainable AI (XAI) will allow developers to understand why a particular bug was detected and how the recommended fix was generated, increasing trust in AI-driven debugging.

1. **Expansion into Security Vulnerability Detection**

* Beyond standard bug detection, future versions of this project could extend to security vulnerability detection, identifying common exploits like SQL injection, buffer overflow, and XSS attacks.

1. **Model Improvement with Reinforcement Learning**

* Using reinforcement learning and continuous model fine-tuning, the system can learn from developer feedback and user interactions to enhance bug detection accuracy over time.
  1. **Conclusion**

The Bug Detection and Fix Recommendation System is a significant step forward in leveraging machine learning and deep learning to automate the debugging process. By detecting potential errors and providing intelligent fix recommendations, this system reduces developer workload, enhances software quality, and speeds up debugging cycles.

While our current implementation provides a strong foundation, there is tremendous potential for future growth, including multi-language support, IDE integration, AI-powered fixes, and security vulnerability detection. With continued research and improvements, this project can become an essential tool for developers, enhancing productivity and ensuring high-quality software development.

1. **References**