**Bug Bot**

**Authors:**

**Aniruddhan S**

Department of Information Technology   
Manipal University Jaipur  
Email: aniruddhan.229302128@muj.mainpal.edu

**Manav Singh**  
Department of Information Technology   
Manipal University Jaipur  
Email: manav.229302140@muj.mainpal.edu

**I.INTRODUCTION**

Coding standards must be followed when developing software in order to guarantee efficiency, readability, and maintainability. Pylint and other code linters help developers spot best practices violations. However, inexperienced programmers may find it difficult to decipher linting errors. In order to improve debugging, this study presents a Python-based code linting tool that incorporates AI-based explanations.

The system lets users choose a Python script, use Pylint to evaluate its quality, and get thorough error explanations using both pre-written descriptions and AI-generated insights. It is made to make learning easier for novice programmers and efficient for more seasoned developers.

**II. SYSTEM ARCHITECTURE**

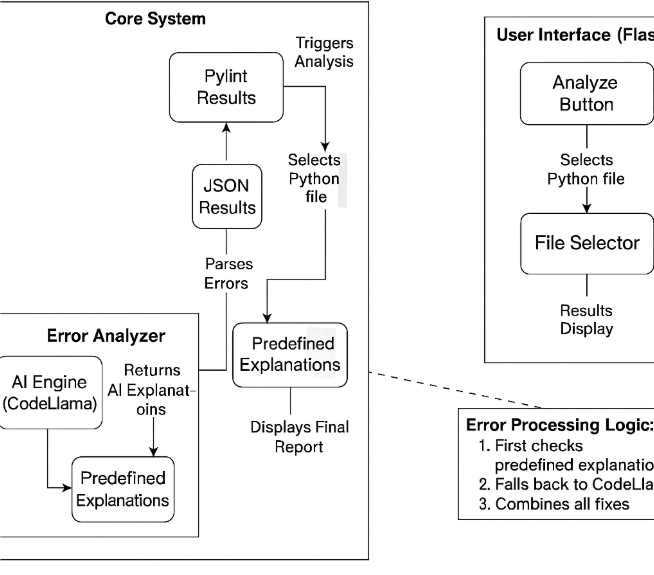
A. **Overview**

The system comprises three major components:

1. **Pylint-based Static Code Analysis:** Finds infractions of coding standards.
2. **Known Error Definition**: Offers standard errors with predetermined messages.
3. **AI-Generated Explanations:** Produces thorough explanations and answers using TinyLlama.

B. **Workflow:**

1. The user chooses a Python file using the GUI.
2. Pylint analyzes the code and produces JSON-formatted results.
3. If errors are found, the system retrieves predefined explanations when they are available.
4. Ollama requests AI-generated explanations for a deeper understanding.
5. The results, along with AI suggestions and error descriptions, are shown in a text area based on a webpage using Flask.



**III. LITERATURE REVIEW**

**1.1 "Tufano et al.'s "Learning Bug-Fixing Patches in the Wild using Neural Machine Translation"**

**Focus**: Learns bug-fix patterns from GitHub Java commits using Neural Machine Translation (NMT).

**Method**: Generates patches by extracting method-level bug-fix pairs (BFPs), abstracting code tokens, and training an encoder-decoder model.

**Key Findings**: 82–99% of generated patches are syntactically valid; 9–50% accuracy in predicting developer-like modifications is achieved.

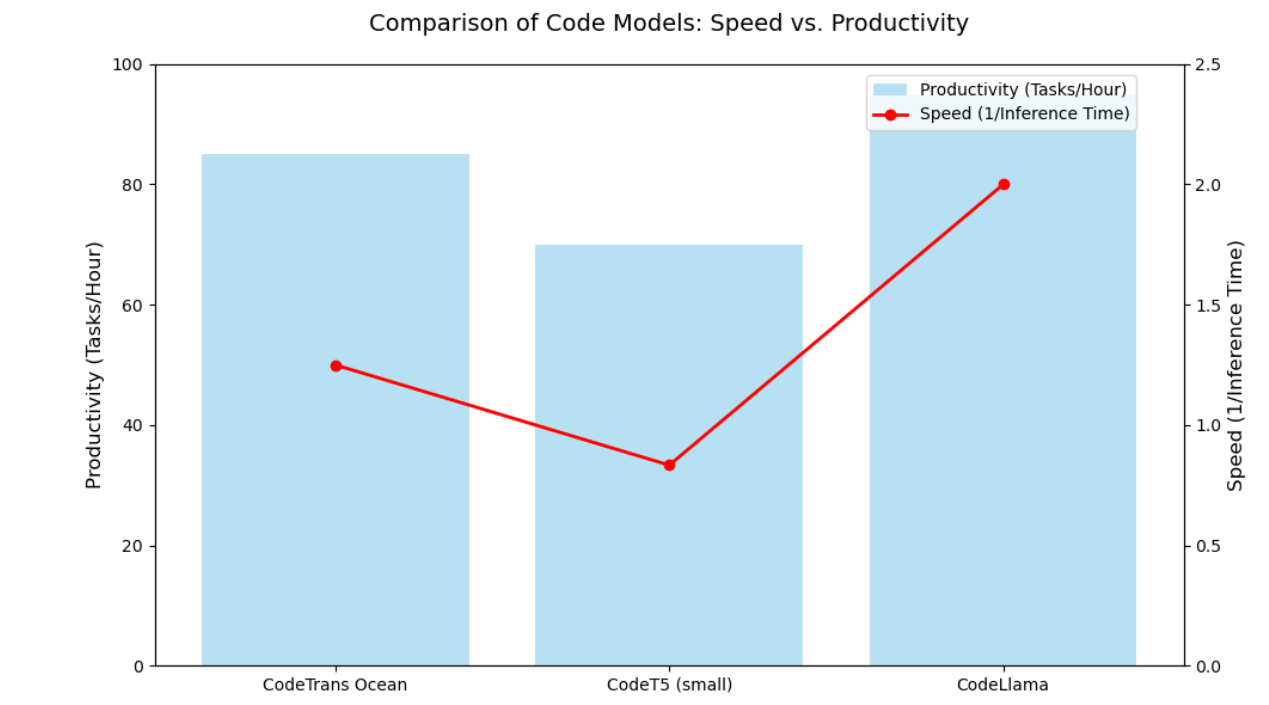
**1.2 (Sejfia et al., ESEC/FSE '21) "Identifying Casualty Changes in Software Patches"**

**Focus**: Introducing casualty changes, which are unnecessary patch adjustments that result from other changes rather than changing the program's logic.

**Method**: Introduces CasCADE, an automated detector that uses AST and static analysis, along with a taxonomy (API-based, variable-based, and refactoring-based).

**Key Findings**: Casualty modifications are present in 21% of security patches. Noise reduction increases the accuracy of patch-based tools (like Assoc Checker) by 18%.

* Our team hasn’t reviewed a lot of research papers due to the fact that there is a deficit of research paper regarding the Problem and its approach to solve the problem.



**III. METHEDOLOGY**

A. **Integration of Pylint** Pylint is run as a subprocess by the run\_pylint\_analysis () function, which also records the output in JSON format.

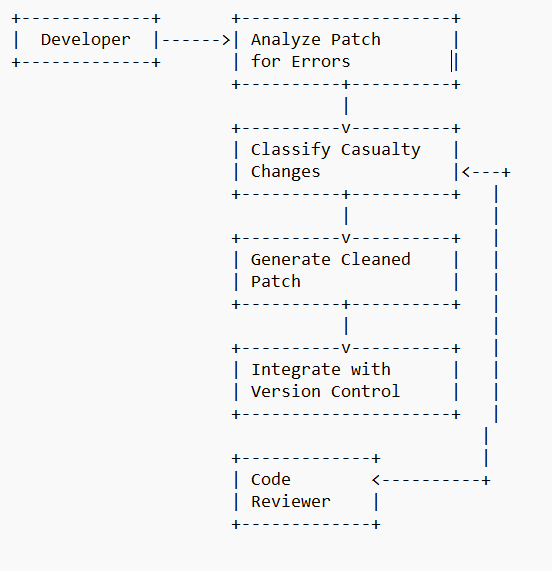
B. **Specified Justifications** Human-readable explanations for frequently occurring errors can be found in a dictionary called PYLINT\_ERROR\_EXPLANATIONS. This makes it possible to give quick feedback without using AI-based processing.

C. **Error Analysis Using AI** The explain\_error\_with\_ollama () function makes use of the CodeLlama model through Ollama for errors that are not addressed by predefined explanations.

D. **User Interface Graphics (GUI)** The application uses a Flask-based web interface that allows users to upload Python files, analyze them, and view results in a clean, styled layout.

Main Components:

1. File Upload Field – Lets users browse and select .py files.
2. Analyze Button – Triggers Pylint analysis and AI-based explanation using TinyLLaMA.
3. Results Display – Shows grouped errors with line numbers and concise AI explanations.
4. History Page – Displays past analyses with timestamps and formatted results.



**IV. RESULTS AND DISCUSSION**

The created solution improves on conventional linting tools by effectively fusing static analysis with AI-generated recommendations. The following advantages are demonstrated by a sample execution:

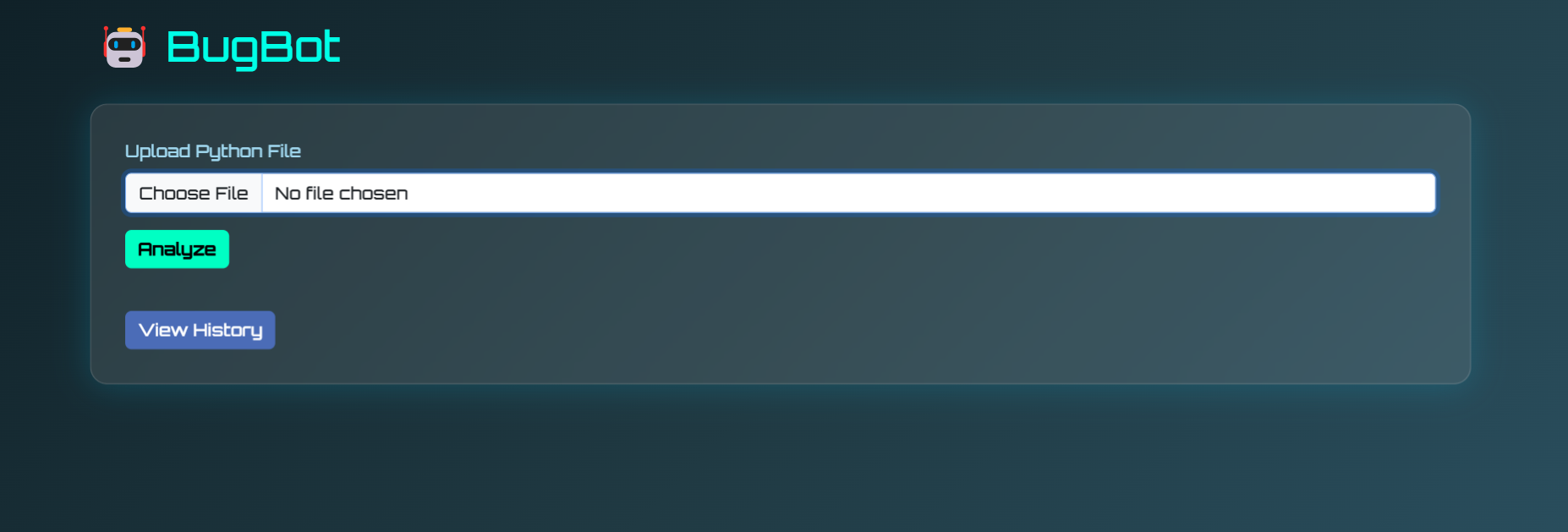
1. **Instant feedback**: Pre-written messages are used to clarify common problems.
2. **Context-aware AI insights**: AI provides thorough justifications for complicated problems.
3. **User-friendly interface** – File selection and result display are made easier in a Webpage.

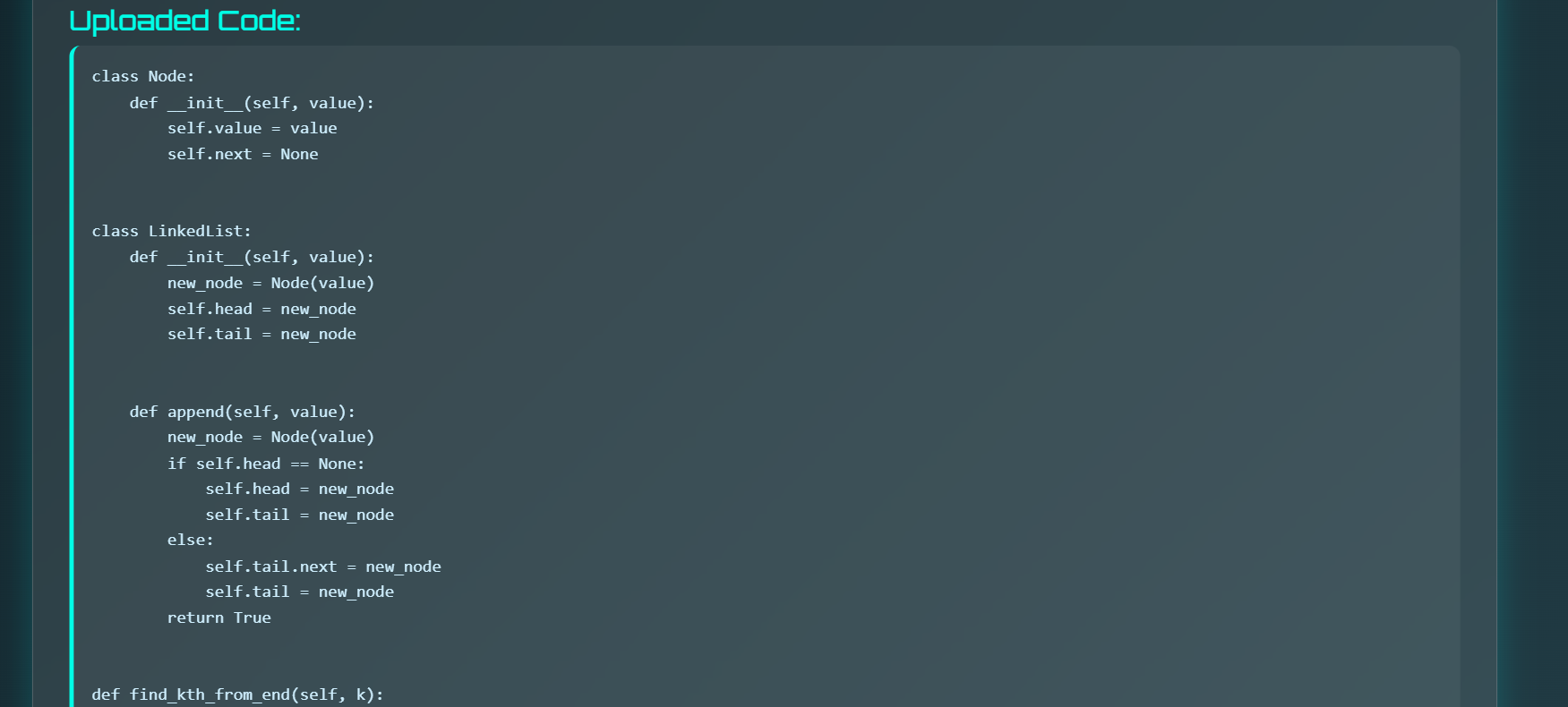
The following are some of the difficulties faced:

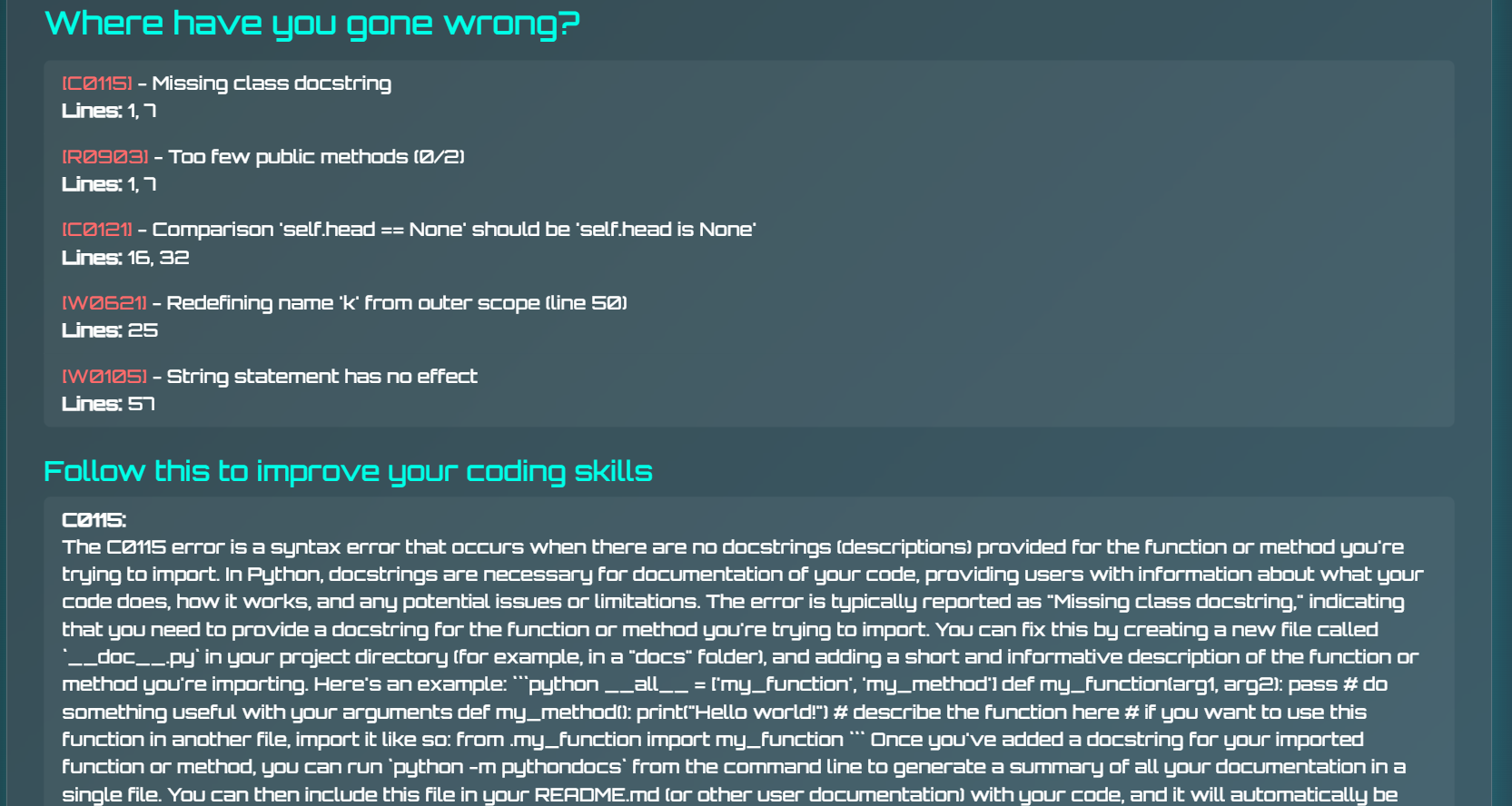
* The response time of AI-generated explanations depends on external API latency.

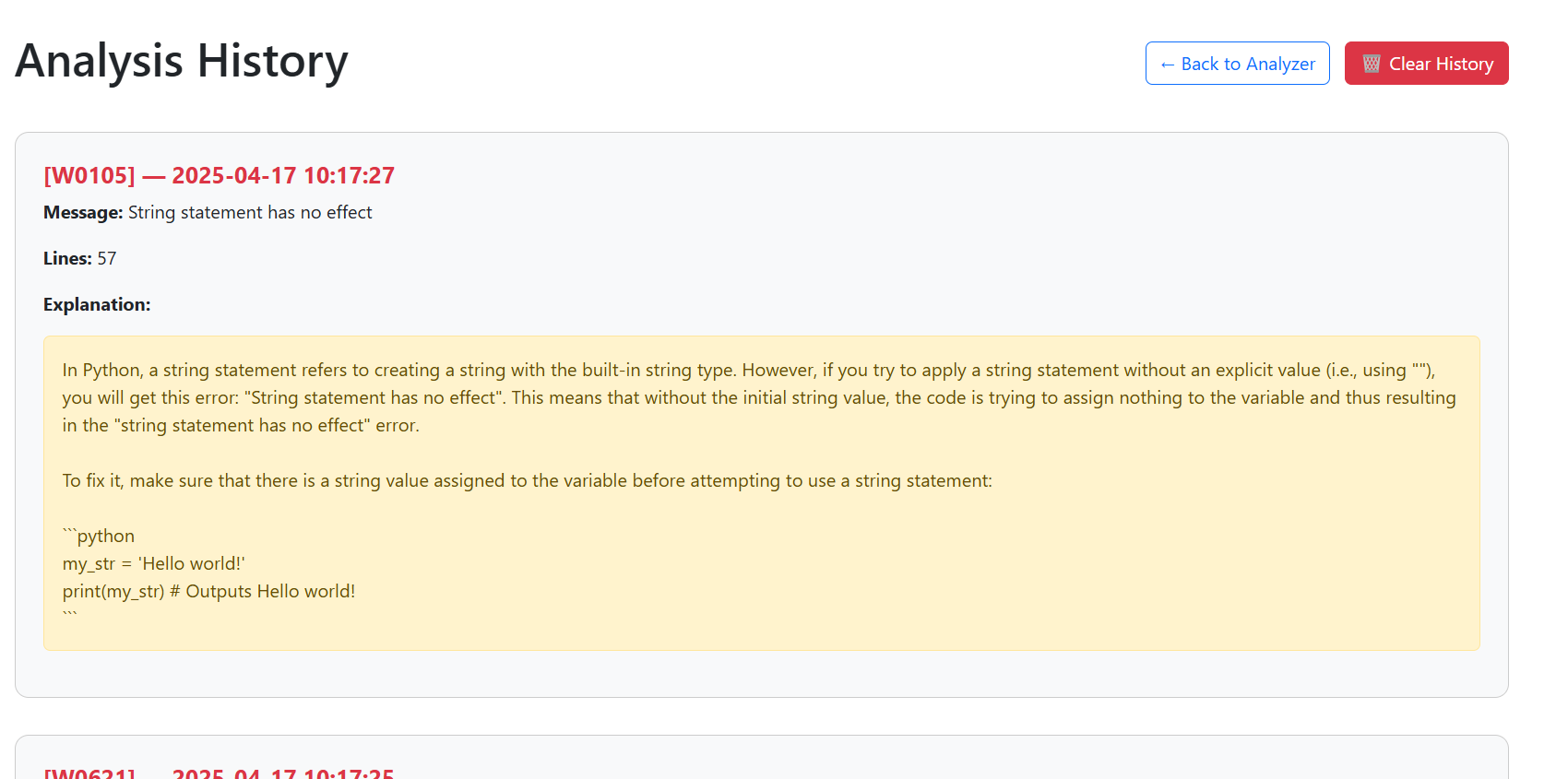
**Results here provided show variations of how Codellama7b can be used:**

* CodeLlama offers different result variations, as you can see below there are 2 different outputs for the same code provided to the LLM.
* The first output provides a code Error ID with a detailed human readable description and also makes sure there is change in code, our project differs from other code linters available because our project makes sure there is change only in the part where an error persists unlike other LLMs which change the entire code compromising the logic which was primarily used to solve the given problem.
* The second approach provides the Error ID alone with a detailed human readable description without a code snippet but does provide a fix in the line/area where the error occurs, this makes the model entirely educational and helps the user solve the problem using his/her own logic without intervening into the process of solving the problem.









**V. CONCLUSION AND FUTURE WORK**

A Python-based linter enhancement tool that incorporates AI-driven explanations for static code inspection is presented in this research. The solution helps developers write cleaner, more maintainable code by bridging the gap between automated mistake detection and human interpretation.

Future enhancements might include:

* Support for more linters (like Flake8 and MyPy) to address a wider variety of problems.
* Code analysis in real time within an extension for an integrated development environment (IDE).
* Better AI model fine-tuning for explanations that are more accurate and contextually sensitive.
* Wire it in with a dedicated LAN and add a proctor computer and make sure that the Proctor computer has access to all the error generation message types and IDs for better teaching and learning experience.

**RESOURCES** [1] "Pylint Documentation," [Online], Python Software Foundation. This link is accessible: https://pylint.pycqa.org/ [2] "Ollama API Documentation," [Online], using OpenAI. Currently accessible: https://ollama.ai/ IEEE Software, vol. 35, no. 6, pp. 32–40, 2020; A. C. Kane, "Improving Software Quality with Static Code Analysis," pp.