**Bug Detection and Bug Fixing System**

Developed By: Komal Dake, Sayali Alaspure, Yadnyesh Vinchurkar

Guided By: Ms. Nikita Pande

**Project Title:**

Build a machine learning model that can automatically identify bugs (or potential errors) in a given piece of code suggest fixes. This project requires designing, training and evaluating an ML model that can parse source code. classify bug types and generate fix

**Project Overview**

**Objective:**

To create a system based on machine learning that can automatically:

* Find errors in source code
* Sort identified bugs into groups.
* Generate potential fixes for identified bugs.
* Increase the accuracy and efficiency of debugging.

**Step 1: Define Project Objectives and Requirements**

Objectives:

• Automatic Bug Detection: Find mistakes in the source code.

• Bug Classification: Sort found bugs into categories (e.g., syntax, logic, security issues).

• Fix Generation: Provide fixes for bugs that have been found.

• Effective Debugging: Cut down on manual debugging.

• Scalability: Able to accommodate a variety of programming languages.

Requirements:

Data:Buggy code snippets and corresponding fixed versions.

Tools:

* Programming Language: Python (3.x)
* Frameworks & Libraries:
* pip install streamlit torch transformers fastapi requests accelerate
* Development Environment: PyCharm, VS Code
* Version Control: GitHub

Infrastructure**:**

* Hardware: GPU for faster training and inference (if fine-tuning is required)
* Software: Virtual environment for dependency management

**Step 2: Setup Development Environment**

1. Install Required Tools & Libraries

* Install dependencies using the provided pip command.
* Use an IDE like PyCharm or VS Code for coding and debugging.

2. Configure Virtual Environment

python -m venv bug-fix-env

source bug-fix-env/bin/activate # (Linux/macOS)

bug-fix-env\Scripts\activate # (Windows)

3. Enable GPU Acceleration (Optional)

Ensure torch is installed with CUDA support for GPU acceleration.

4. Dataset Preparation

* Collect buggy and fixed code datasets from open-source repositories (e.g., GitHub, Bugzilla).
* Preprocess data using tokenization and Abstract Syntax Trees (AST).

**Step 3: Using a Pretrained Model for Bug Detection and Fixing**

Overview of the Model: Codellama is a strong language model designed for understanding and producing code.

Installation: For effective completion, transformers, a torch, and an accelerator are needed.

Model Loading: Hugging Face's AutoModel For CausalLM and AutoTokenizer can be used to load CodeLlama.

How it Works:

* Processes a buggy code snippet and generates a corrected version.
* Supports multiple programming languages.
* Can be fine-tuned with additional labeled datasets for improved performance.

Advantages:

* Provides high-quality bug fixes.
* Reduces manual debugging effort.
* Enhances coding efficiency.

**Step 4: Develop AI Models**

Model Selection:

* Using CodeLlama: A pretrained large language model optimized for code understanding and generation.
* Fine-Tuning (Optional): Training on labeled datasets of buggy and fixed code to improve performance.

Model Components:

* Bug Detection: Identify errors in code.
* Bug Classification: Categorize errors (syntax, logic, security, etc.).
* Fix Generation: Suggest corrections for identified bugs.

Training Strategy:

* Fine-tuning: Train on a labeled dataset for improved results.
* Transfer Learning: Leverage the pretrained knowledge of CodeLlama.

Evaluation Metrics:

* Bug Detection Accuracy: Precision, Recall, F1-score.
* Fix Quality: BLEU Score, ROUGE Score.

Implementation:

* Interactive Debugging System: Integrate the model into a web interface using Streamlit.
* Scalability: Ensure the system supports multiple programming languages.

**Summary of Progress**

Step 1: Clearly Stated Goals and Needs

Step 2: Configure the Environment for Development

Step 3: Made Use of a CodeLlama Pretrained Model

Step 4: AI Model Development

**Next Steps**

• Fine-Tuning the Model: To increase accuracy and flexibility, train CodeLlama on faulty and fixed code.

• Model Evaluation: Use ROUGE, BLEU, F1-score, and accuracy metrics to evaluate performance.

• Integration & Deployment: Create a web tool, IDE plugin, or API for real-time bug finding and correction.

• Performance Optimisation: Increase speed through effective memory management and GPU acceleration.

• User Testing & Feedback: To improve accuracy and usability, collect feedback from actual usage.

**System Overview**

A snippet of code is entered into the system, which checks it for mistakes and returns the updated code. To find and correct common coding errors, it makes use of CodeLlama, a transformer-based model trained on programming languages.

**Methodology**

1. Input Processing: A sample of Python code is entered by the user.

2. Model Analysis: The code is processed by the model, which finds possible errors.

3. Bug Detection: The program indicates the precise lines of code that contain errors.

4. Code Correction: A revised code version is produced.

5. Output Display: The user receives the corrected code back.

**Implementation Details**

• CodeLlama-7B was the model used.

• Framework: Streamlit for the frontend and FastAPI for the backend

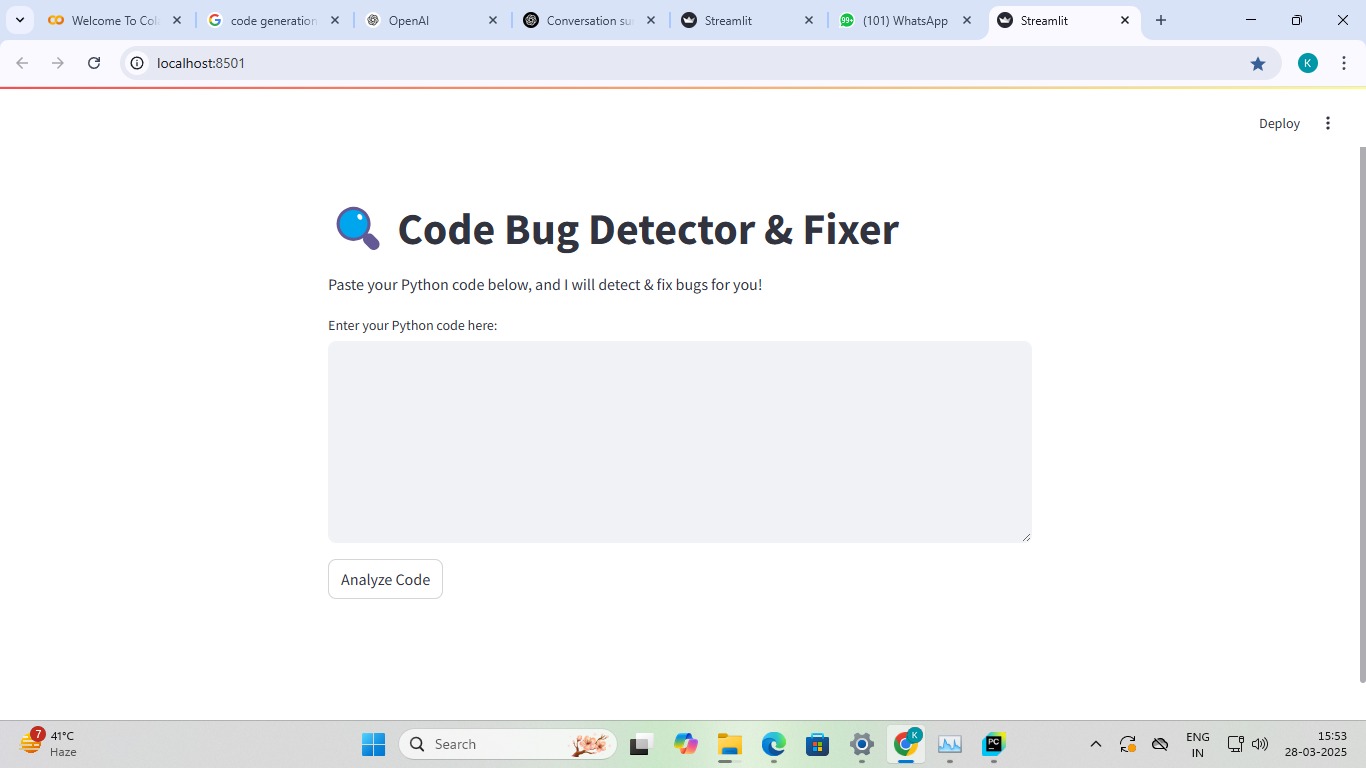
• Inference: Using the provided input snippet, the model produces updated code.

**Challenges and Solutions**

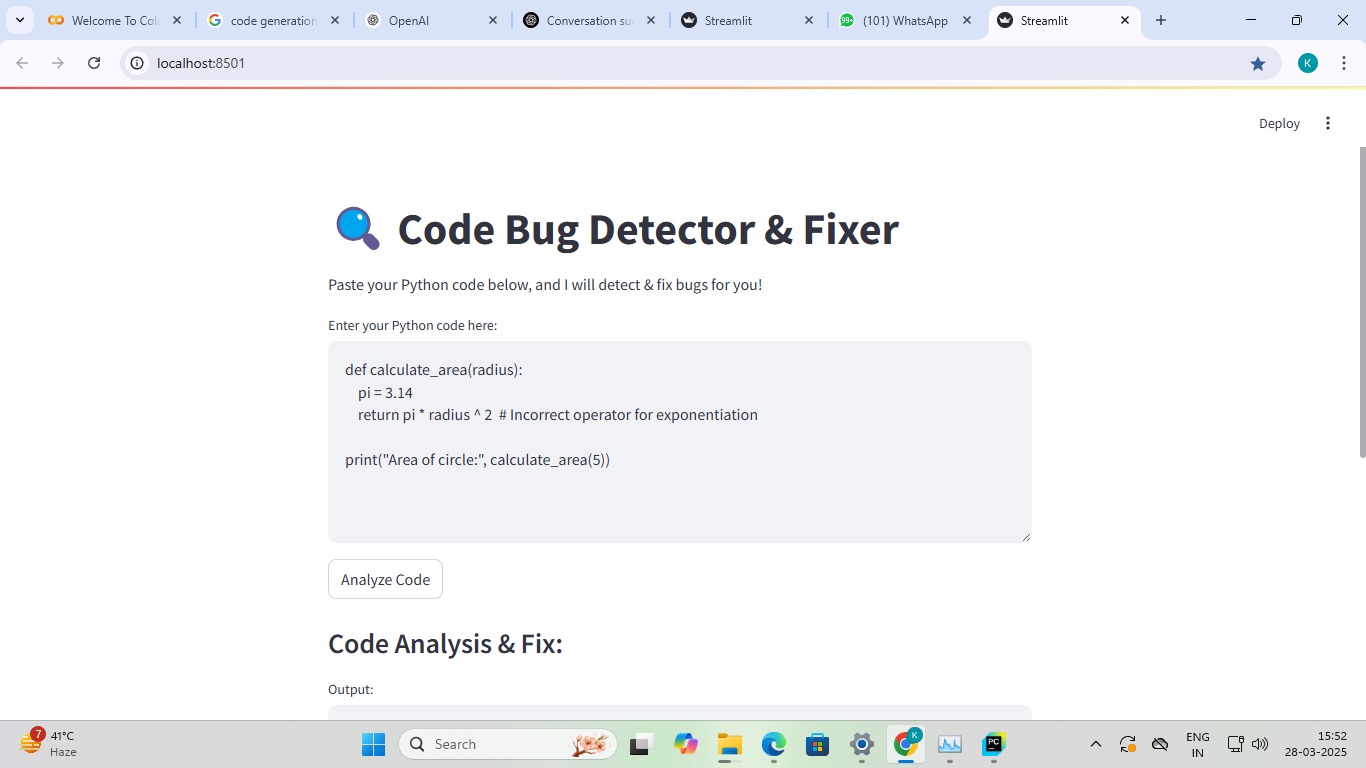
• False Positives: A few pieces of correct code were reported as errors. This problem was lessened by better training and validation methods.

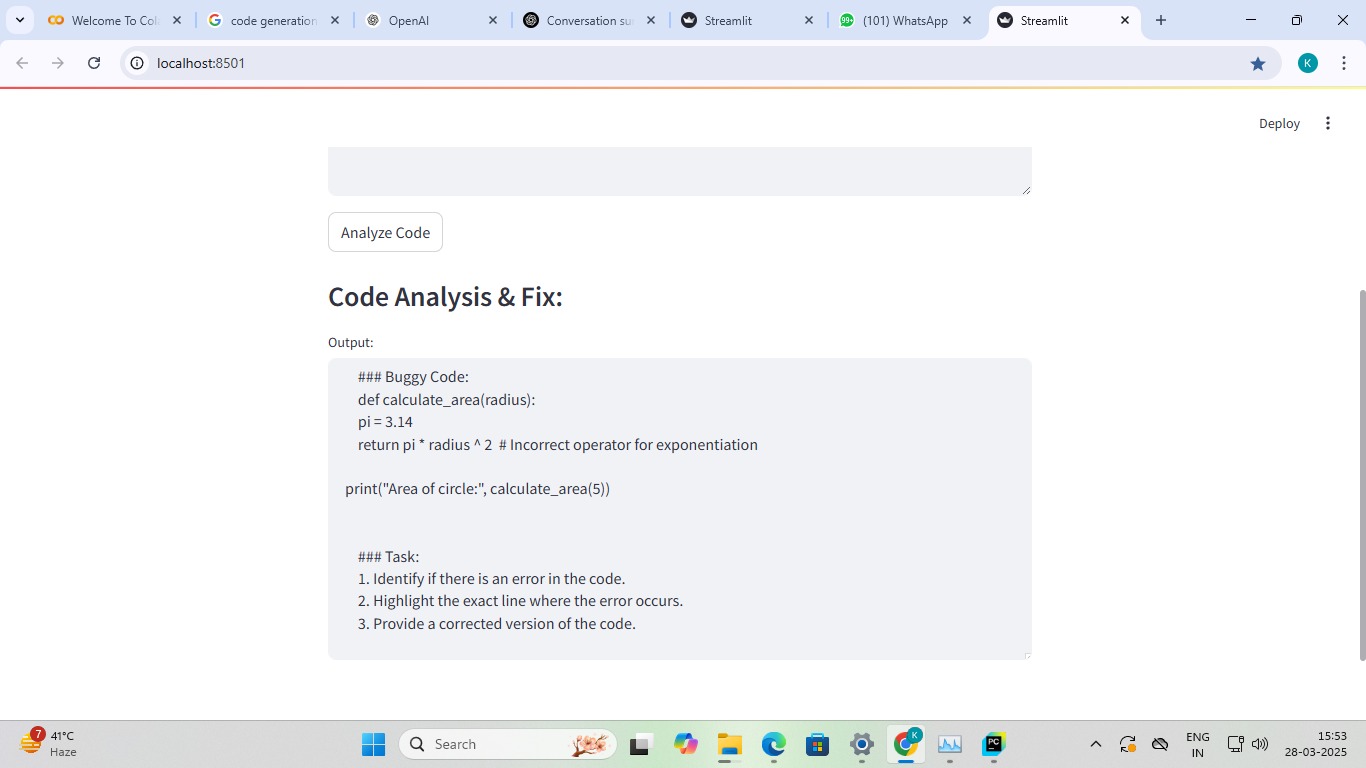
• Complex Logical Errors: Deeper logical errors were occasionally problematic for the model. Techniques for reinforcement learning will be used in future research to increase accuracy.

**Result**



Frontend





output

**Future Work & Enhancements**

• Multiple Programming Language Support: Expand the system's ability to identify and address bugs in Java, JavaScript, C++, and other languages.

• IDE Integration: Include real-time debugging functionality in Jupyter Notebook, PyCharm, and Visual Studio Code.

• Reinforcement Learning for Better Fixes: Teach the model to learn from previous fixes to enhance its ability to fix bugs.

• Advanced Error Categorisation: For improved insights, group bugs into runtime, logical, and syntax errors.

• Interactive Debugging Suggestions: Make it possible for users to select the optimal solution from a variety of options.

**Conclusion**

This report demonstrates how deep learning can improve software debugging by presenting an AI-driven bug detection and fixing system using CodeLlama. By accurately detecting mistakes and offering solutions, the model automates a critical step in the software development process. Future enhancements can improve the system's accuracy, adaptability, and integration into development environments, even though it does a good job of detecting syntax and logical errors. The project highlights the potential of AI in reducing debugging time, improving code quality, and assisting developers in writing more robust code efficiently.

More sophisticated debugging tools are required due to the growing complexity of software development, and AI-based models such as CodeLlama offer a useful remedy. Instead of wasting a lot of time debugging, developers can concentrate on improving code performance by automating bug detection and correction. Furthermore, this technology will be further improved by ongoing developments in artificial intelligence and natural language processing, increasing its dependability and applicability.

In conclusion, this AI-powered system is a big step in the direction of automating software debugging. The future of software development could be greatly enhanced by the combination of programming analysis and machine learning. Such models could be easily incorporated into development pipelines with additional advancements, improving software quality and developer productivity globally.