A Project Report on

"Deep Learning Approach for Automated Bug Detection and Fix Recommendation"

Submitted to

DR. BABASAHEB AMBEDKAR TECHNOLOGICAL UNIVERSITY, LONERE

In fulfillment of the requirement for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

By

Takbide Shubhangi Madhav

Kubade Shruti Suresh

Gorgile Shruti Dnyanoba

Under the Guidance

of

Ms. N. L. Pariyal

(Department of Computer Science and Engineering)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MAHATMA GANDHI MISSION'S COLLEGE OF ENGINEERING,

NANDED(M.S.)

Academic Year 2024-25

Certificate

This is to certify that the project entitled

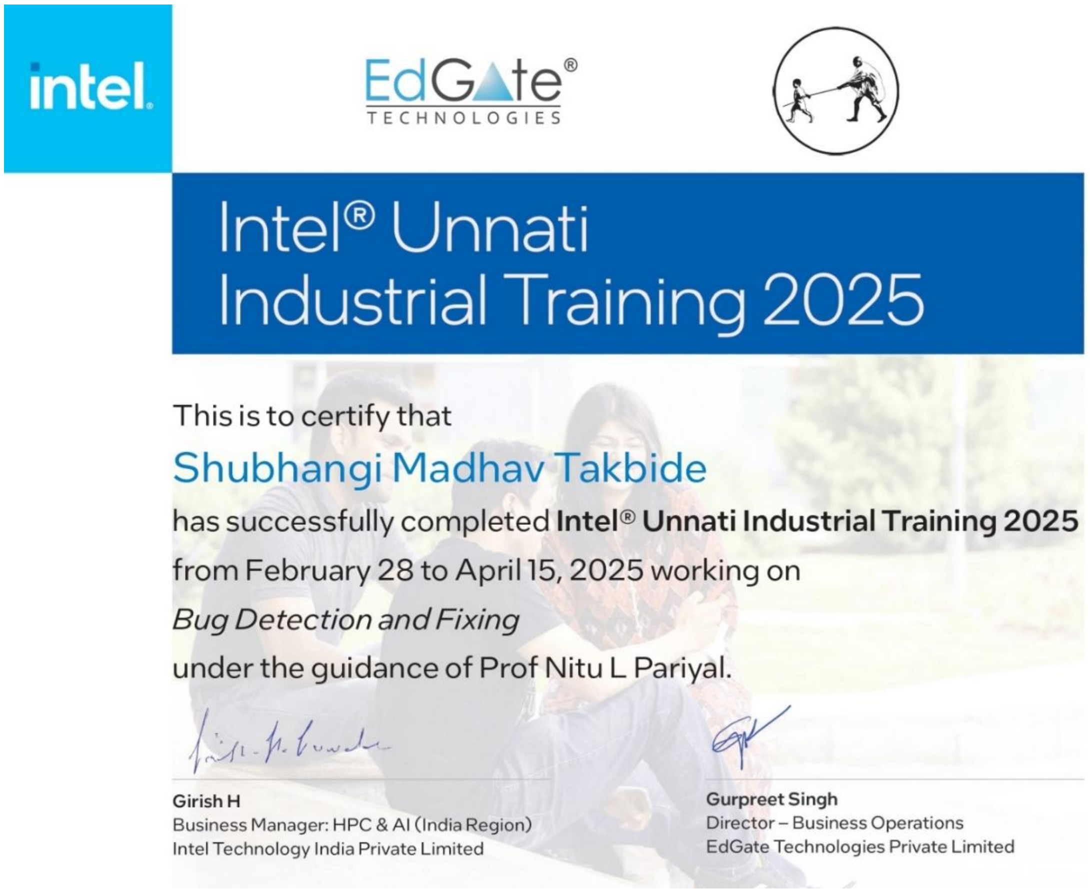
"Deep Learning Approach for Automated Bug Detection and Fix Recommendation"

being submitted by ***Miss. Takbide Shubhangi, Kubade Shruti, Gorgile Shruti***
to the Dr. Babasaheb Ambedkar Technological University, Lonere , for the award of the degree of Bachelor of Technology in Computer Science and Engineering, is a record of bonafide work carried out by them under my supervision and guidance. The matter contained in this report has not been submitted to any other university or institute for the award of any degree.

Ms. N. L. Pariyal

Project Guide

Dr. A. M. Rajurkar	**Dr. G. S. Lathkar**
H.O.D	**Director**
Computer Science & Engineering	MGM's College of Engg., Nanded



intel.

EdGate® TECHNOLOGIES

AICTE

Intel® Unnati Industrial Training 2025

This is to certify that

Shubhangi Madhav Takbide

has successfully completed **Intel® Unnati Industrial Training 2025** from February 28 to April 15, 2025 working on

Bug Detection and Fixing

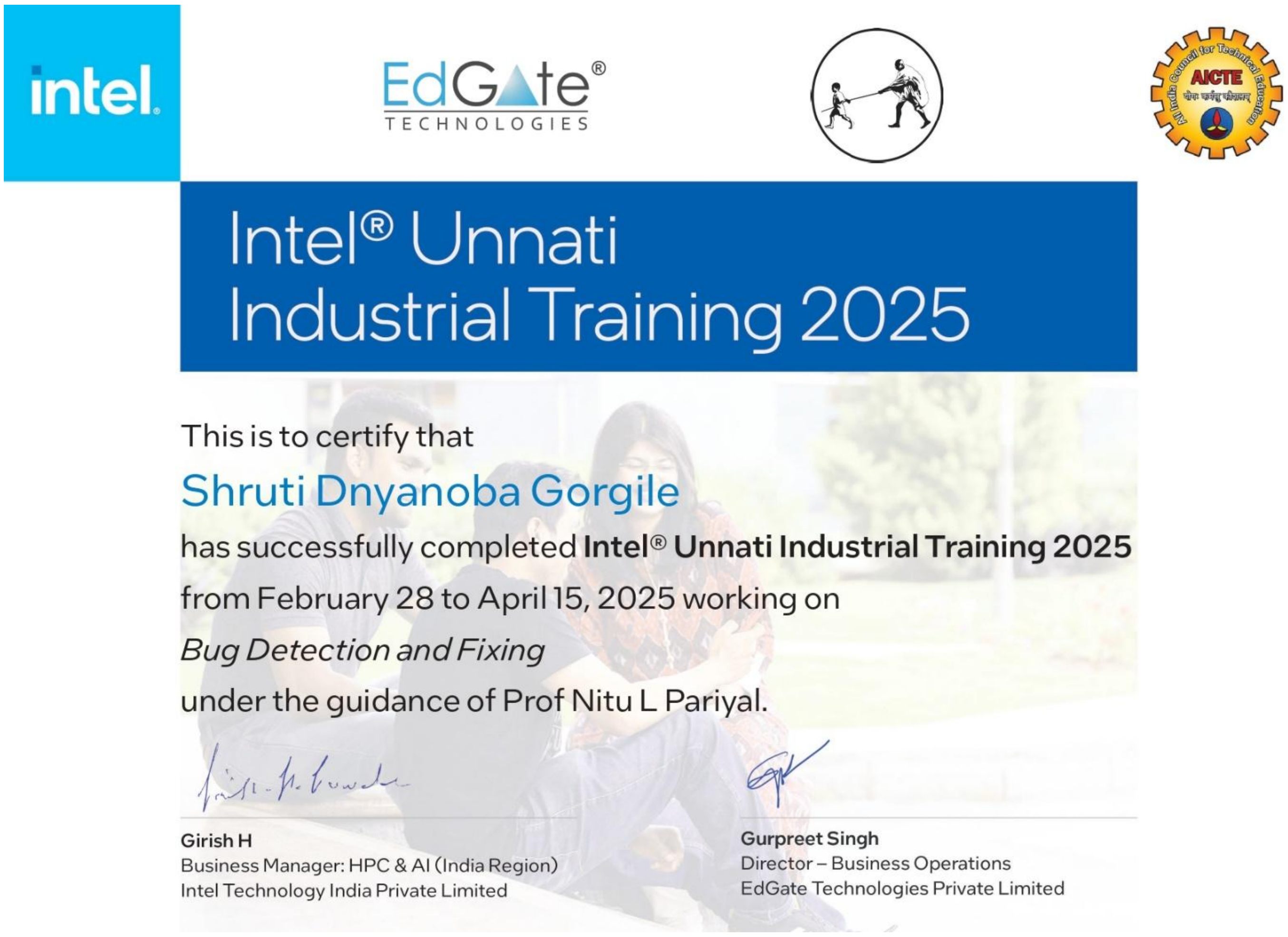
under the guidance of Prof Nitu L Pariyal.

[illegible]

Girish H
Business Manager: HPC & AI (India Region)
Intel Technology India Private Limited

[illegible]

Gurpreet Singh
Director – Business Operations
EdGate Technologies Private Limited



Intel® Unnati Industrial Training 2025

This is to certify that

Shruti Dnyanoba Gorgile

has successfully completed **Intel® Unnati Industrial Training 2025** from February 28 to April 15, 2025 working on

Bug Detection and Fixing

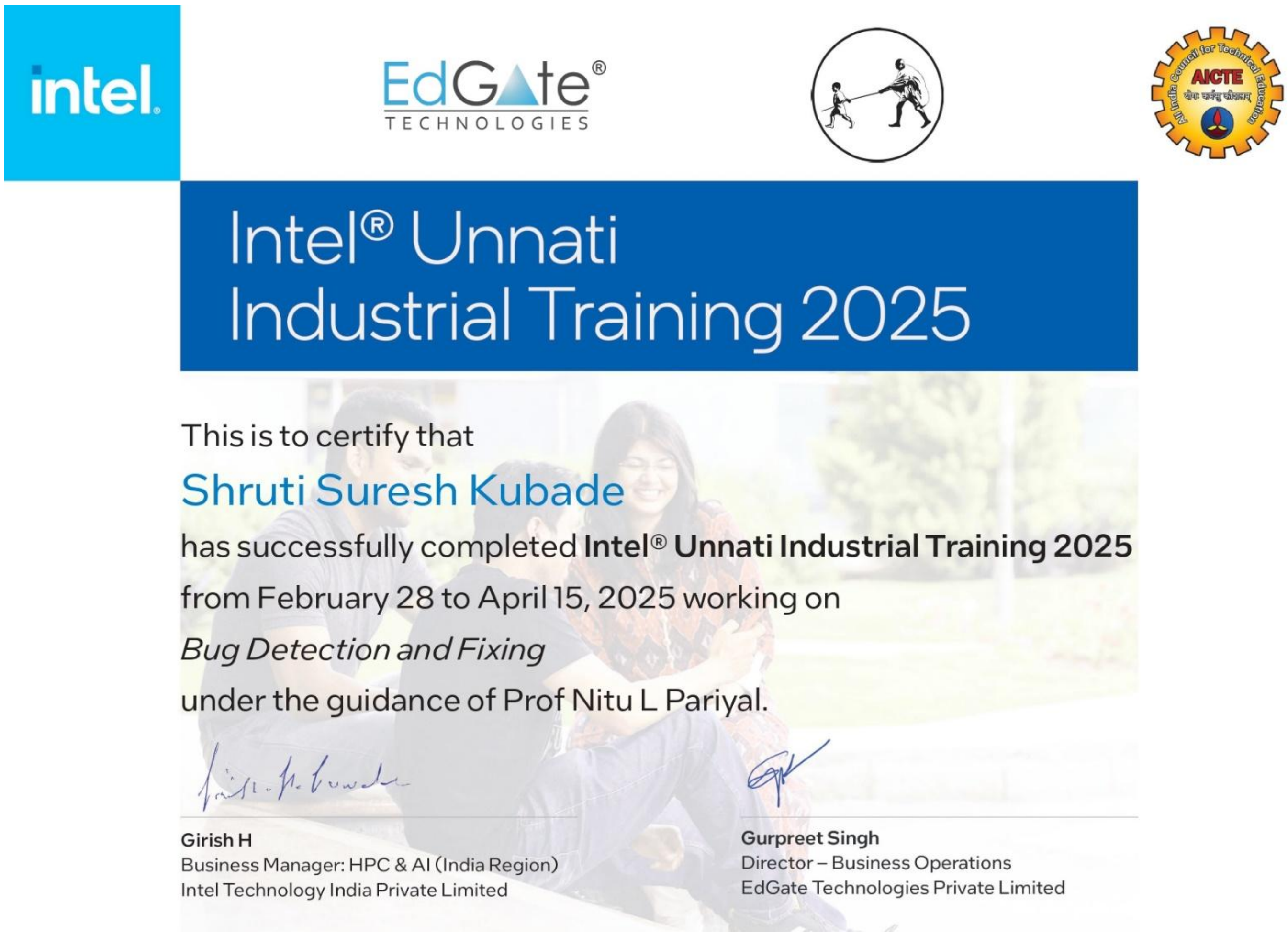
under the guidance of Prof Nitu L Pariyal.

[illegible]

Girish H
Business Manager: HPC & AI (India Region)
Intel Technology India Private Limited

[illegible]

Gurpreet Singh
Director – Business Operations
EdGate Technologies Private Limited



Intel® Unnati Industrial Training 2025

This is to certify that

Shruti Suresh Kubade

has successfully completed **Intel® Unnati Industrial Training 2025** from February 28 to April 15, 2025 working on *Bug Detection and Fixing* under the guidance of Prof Nitu L Pariyal.

[illegible]

Girish H
Business Manager: HPC & AI (India Region)
Intel Technology India Private Limited

[illegible]

Gurpreet Singh
Director – Business Operations
EdGate Technologies Private Limited

ACKNOWLEGEMENT

We would like to express my deepest gratitude to my project guide, **Ms. N. L. Pariyal**, for her invaluable support, guidance, and encouragement throughout this project. Her profound knowledge and expertise have been instrumental in the successful completion of this work. Her patience and willingness to assist us at every step have greatly enriched our learning experience. Her constructive feedback and insightful suggestions have not only helped us overcome challenges but also motivated us to strive for excellence.

We gladly take this opportunity to thank **Dr. A. M. Rajurkar** (Head of Computer Science & Engineering, MGM's College of Engineering, Nanded). We are heartily thankful to **Dr. G. S. Lathkar** (Director, MGM's College of Engineering, Nanded) for providing facilities during the progress of the project and also for her kind help, guidance and inspiration. Last but not least, we are also thankful to all those who helped, directly or indirectly, develop this project and complete it successfully.

With Deep Reverence,

Takbide Shubhangi

Kubade Shruti

Gorgile Shruti

B.TECH [CSE]

ABSTRACT

Bugs remain a significant bottleneck in modern software development, impacting system performance, maintainability, security, and reliability. These issues can arise from various sources, including syntax errors, logical flaws, and runtime failures. Traditional debugging methods—such as manual testing, static code analysis, and runtime tracing—are often slow, error-prone, and heavily reliant on developer expertise.

This project presents an AI-driven automated debugging system capable of detecting, classifying, localizing, and fixing bugs in Python code. At the core of the system is a fine-tuned version of CodeLlama, a transformer-based large language model specialized in code understanding and generation. Rather than utilizing the base model as-is, we customized it using a curated dataset composed of buggy and corrected Python code pairs. This dataset captures a wide spectrum of real-world bug types, including syntax mistakes, indentation issues, logical inconsistencies, and runtime errors. The system operates in four sequential stages. First, it performs bug detection to identify the presence of any code issues. Next, it classifies the nature of the bug, distinguishing between syntax, logic, and semantic errors. following classification, the system localizes the bug by identifying the specific line or lines of code responsible. Finally, it applies a fix, generating a corrected version of the input code.

Unlike conventional debugging tools, this system leverages the contextual reasoning capabilities of transformer models to understand and correct code based on its semantics and structure. It has been rigorously tested on both synthetic datasets and real-world Python scripts, consistently demonstrating high accuracy in detecting and resolving common bugs with minimal developer intervention. This work highlights the effectiveness of adapting large language models to specialized software engineering tasks. By automating key aspects of the debugging process, the system reduces developer workload and enhances code quality. Future improvements will focus on expanding the training dataset, improving the handling of complex multi-line bugs, and extending support to additional programming languages beyond Python.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

In recent years, the complexity of software systems has increased exponentially, resulting in greater difficulty in maintaining and debugging code. Bugs—whether caused by human error, lack of testing, or logic faults—can severely impact the performance, security, and reliability of applications. For developers, especially beginners, identifying and fixing these bugs manually is a tedious and time-consuming process. As software development timelines become shorter and expectations for quality increase, the need for intelligent, automated debugging solutions becomes more critical.

Artificial Intelligence (AI) and Machine Learning (ML), especially in the form of large language models like CodeLlama, have demonstrated impressive capabilities in understanding and generating source code. These models can be leveraged not only for code completion but also for more advanced tasks such as bug detection, code repair, and classification. This project explores how such AI models can be fine-tuned and applied to solve real-world coding errors in Python automatically.

1.1 Overview of the Project

Software bugs are one of the most common and critical challenges in the software development process. Bugs can occur due to syntax errors, logical flaws, incorrect variable handling, or improper flow control. They can cause a program to behave incorrectly, crash unexpectedly, or produce inaccurate results. As codebases grow larger and more complex, identifying and fixing these bugs becomes increasingly difficult.

Conventional debugging techniques, such as manual code reviews, print statements, and testing frameworks, often require significant developer time and effort. These methods are also prone to human error and cannot guarantee full coverage of possible bugs. In recent years, the emergence of Artificial Intelligence (AI) and Machine

Learning (ML), especially transformer-based language models like CodeLlama, has opened up new opportunities for intelligent and automated code analysis. This project proposes a system for automatic bug detection, classification, line-level identification, and correction in Python code using a fine-tuned version of the CodeLlama model. Unlike general-purpose usage, the CodeLlama model used in this project is fine-tuned using a custom dataset of buggy and corrected Python code snippets, specifically designed and labeled for this task. The system can process source code input, detect whether it contains bugs, identify the type of bugs, locate the bug lines, and suggest or directly apply fixes.

1.2 Problem Statement

Debugging is one of the most time-consuming and labor-intensive phases of the software development lifecycle. It requires strong domain knowledge, experience, and constant vigilance. In large-scale software systems, it is often difficult to pinpoint the source of a bug, especially when it involves subtle logical or semantic issues. While static analysis tools and linters can help detect some issues, they cannot fix the problems or understand the broader context of the code. Additionally, most current solutions focus on either detection or fixing, but not both.

There is a need for an end-to-end automated system that:

- Detects bugs
- Classifies them by type
- Identifies their location
- Suggests or applies fixes

1.3 Objectives

The primary aim of this final year engineering project is to design and implement an intelligent, automated system capable of detecting and fixing bugs in Python source code. This initiative focuses on leveraging advancements in machine learning and large language models to enhance software reliability and streamline the debugging process. The following objectives guide the overall development and evaluation of the system:

Firstly, the project begins with the construction of a custom dataset tailored specifically for the task of bug detection and correction. This dataset comprises pairs of buggy Python code and their corrected counterparts. The purpose of creating such a dataset is to provide the model with concrete examples of common programming mistakes and their resolutions. This curated data serves as the foundation for training and testing the model's capabilities. The second objective involves the fine-tuning of the CodeLlama model**,** a powerful language model specialized in code generation and understanding. Fine-tuning on our custom dataset aims to adapt the pre-trained model more closely to the specific task of Python bug detection and correction. By doing so, the model becomes more proficient in recognizing patterns of errors and proposing appropriate fixes within the Python programming context.

Once trained, the model is expected to perform three core functions autonomously. The first is bug detection, where the system analyzes a given Python code snippet to determine whether it contains any bugs. This requires a deep semantic and syntactic understanding of the code to accurately judge its correctness. Following detection, the model proceeds to classify the type of bug identified. The system is designed to distinguish between various bug categories such as syntax errors (violations of Python grammar rules), logical errors (incorrect implementation of intended logic), and runtime issues (errors that occur during code execution). This classification is crucial for prioritizing and applying appropriate corrective strategies.

Next, the model focuses on bug localization, which entails pinpointing the exact line or lines of code responsible for the error. Accurate localization greatly enhances the efficiency of debugging by narrowing down the search area for the developer or automated fixing system. Building on localization, the system then performs automated bug fixing. Using the insights gathered, the fine-tuned model generates corrected versions of the buggy code. The aim is to produce solutions that are not only syntactically valid but also functionally accurate, preserving the original intent of the program. Lastly, to validate the effectiveness of the system, a comprehensive evaluation process is carried out using real-world Python code samples. This involves testing the system's ability to detect and fix bugs in practical scenarios and measuring key performance metrics such as accuracy, precision, recall, and overall reliability. The results from this evaluation help in assessing the practical utility and robustness of the proposed solution.

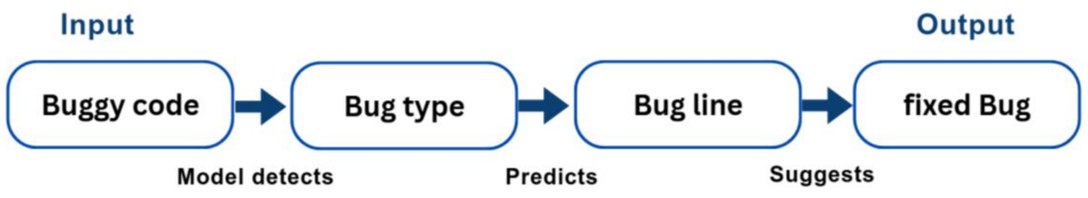


Fig. 1.1: Project objectives work flow

1.4 Scope of the Project

The scope of this project is precisely defined to ensure focused and achievable outcomes within the constraints of a final year engineering endeavor. The core functionality is centered on automated debugging of Python source code, leveraging modern machine learning techniques and pre-trained large language models. Python is chosen due to its widespread use, readable syntax, and the abundance of available resources, making it an ideal candidate for such research and application.

The system is designed to address and correct three major categories of programming bugs:

1. Syntax Errors: These include common mistakes such as missing colons, improper indentation, unmatched brackets or parentheses, and other violations of Python's grammatical structure. These are often the first barrier to program execution and can usually be identified through static code analysis.
2. Logical Errors: These bugs occur when the code runs without syntax issues but does not produce the expected result. Examples include incorrect loop boundaries, flawed conditionals, or improper use of control structures. Logical errors are often the hardest to detect as they require understanding the programmer's intent.
3. Semantic Issues: These are related to the meaning and context of the code, such as the use of undefined variables, mismatched data types, or incorrect function

arguments. Such bugs often lead to runtime errors or incorrect program output, necessitating deeper analysis of code behavior.

To support these functionalities, the project includes several core components:

- Dataset Creation and Curation: A significant part of the project involves generating or collecting a high-quality labeled dataset of buggy and corrected Python code snippets. The dataset must cover a diverse range of error types and code structures to effectively train the model.
- Model Fine-Tuning: The CodeLlama model, a large language model specialized in code tasks, is fine-tuned using the curated dataset. Efficient tuning strategies are employed, such as parameter-efficient transfer learning, to adapt the model without requiring extensive computational resources.
- Interactive Interface Development: A simple yet functional user interface or script is developed to allow users to input their Python code. The system then processes this input to detect bugs, provide classification, and return the corrected code. This component enhances the usability and accessibility of the tool for non-expert users.

It is important to acknowledge the limitations of the current scope. This version of the system does not extend support to multi-language code debugging (e.g., Java, C++), deep runtime analysis (such as dynamic tracing or memory profiling), or integration with professional development environments (IDEs). These advanced features are reserved for future extensions of the project, which could involve integrating runtime behavior analysis, cross-language compatibility, or embedding the system as a plugin within IDEs for real-time feedback. In conclusion, the scope of this project is purposefully limited to ensure depth and quality in addressing the targeted problem domain: Python code bug detection and correction. The foundational work laid out in this project establishes a platform that can be further expanded and enhanced in future research or industrial applications.

1.5 Significance of the Study

The proposed project holds substantial significance in the fields of software engineering, artificial intelligence, and computer science education. By combining machine learning techniques with automated debugging, it brings forth a transformative approach to how developers interact with and improve their code. Below is an in-depth elaboration of the key points that highlight the importance and impact of this study:

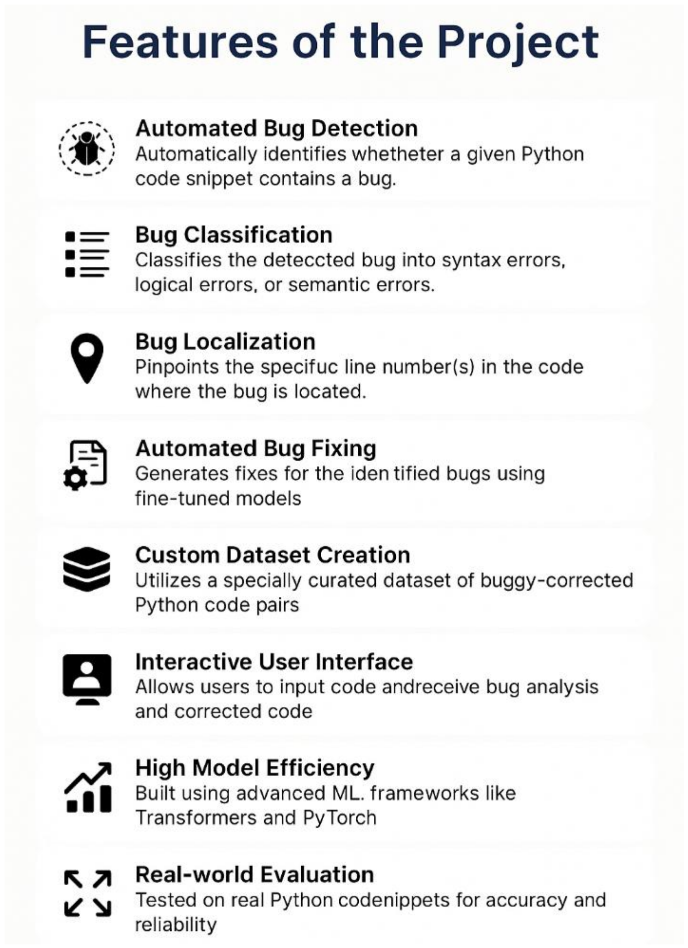
Firstly, the project introduces an automated and intelligent alternative to traditional debugging practices. Manual debugging is often time-consuming, prone to oversight, and can become a bottleneck in the development lifecycle. This system offers a more efficient, AI-driven solution that can analyze code and suggest corrections in a fraction of the time it would take a human, thereby streamlining the software development process.

Secondly, the system reduces development time by automating the processes of bug identification and correction. Developers frequently spend a significant portion of their workflow identifying and fixing bugs, especially subtle logical or semantic errors that evade immediate notice. An automated tool that can quickly flag and fix such issues allows developers to focus more on implementing features and optimizing performance, ultimately increasing productivity.

Thirdly, the project holds particular value for beginner and novice programmers. Learning to code can be overwhelming, especially when students encounter bugs without understanding their root causes. This tool provides immediate, contextual feedback, helping learners grasp common mistakes and learn from corrections. It acts as an educational assistant, making programming more approachable and less frustrating.

Moreover, the project showcases the practical application of large language models like CodeLlama in real-world software engineering tasks. While these models are often used for code generation or completion, this study demonstrates their extended capability in code understanding, classification, and error correction. This contributes to the growing body of research on LLMs in code intelligence.

As illustrated in Figure 2.7, the project incorporates a wide range of features, from automated bug identification and correction to dataset creation and real-world evaluation, making it a comprehensive tool for Python code debugging.



Features of the Project

Automated Bug Detection
Automatically identifies whetheter a given Python code snippet contains a bug.

Bug Classification
Classifies the deteccted bug into syntax errors, logical errors, or semantic errors.

Bug Localization
Pinpoints the specifuc line number(s) in the code where the bug is located.

Automated Bug Fixing
Generates fixes for the iden tified bugs using fine-tuned models

Custom Dataset Creation
Utilizes a specially curated dataset of buggy-corrected Python code pairs

Interactive User Interface
Allows users to input code andreceive bug analysis and corrected code

High Model Efficiency
Built using advanced ML. frameworks like Transformers and PyTorch

Real-world Evaluation
Tested on real Python codenippets for accuracy and reliability

Fig. 1.2: Features of the AI-Based Bug Detection and Fixing System

Additionally, the work contributes a reusable and valuable dataset of buggy and corrected Python code. This dataset, along with the methodology developed for training and evaluating the model, can serve as a foundation for future projects and research in similar domains. It opens the door for replication, enhancement, and benchmarking in the AI-assisted programming community. The system also has potential for adaptation and scaling in professional development environments and educational platforms. With further refinement, it could be integrated into IDEs, online coding platforms, or interactive learning tools, offering real-time code analysis and support for learners and professionals alike.

Finally, this project lays the groundwork for future innovation in AI-assisted software development. It is an early step toward creating autonomous programming assistants that not only generate code but also ensure its correctness. The approach demonstrated here could evolve into more advanced tools capable of understanding complex software systems and collaborating with developers as intelligent coding partners.

1.6 Existing Technologies and Solutions

To address the problem of inefficient and manual debugging, several technologies have emerged over the years. Traditional tools like static analyzers, linters, and IDE extensions (e.g., PyLint, SonarQube, VS Code IntelliSense) can detect basic issues such as syntax errors, unused variables, or simple formatting problems. While helpful, these tools rely heavily on predefined rules and struggle to detect deeper issues like logic or runtime errors. Most importantly, they don't provide automated code fixes or understand the broader intent behind the code.

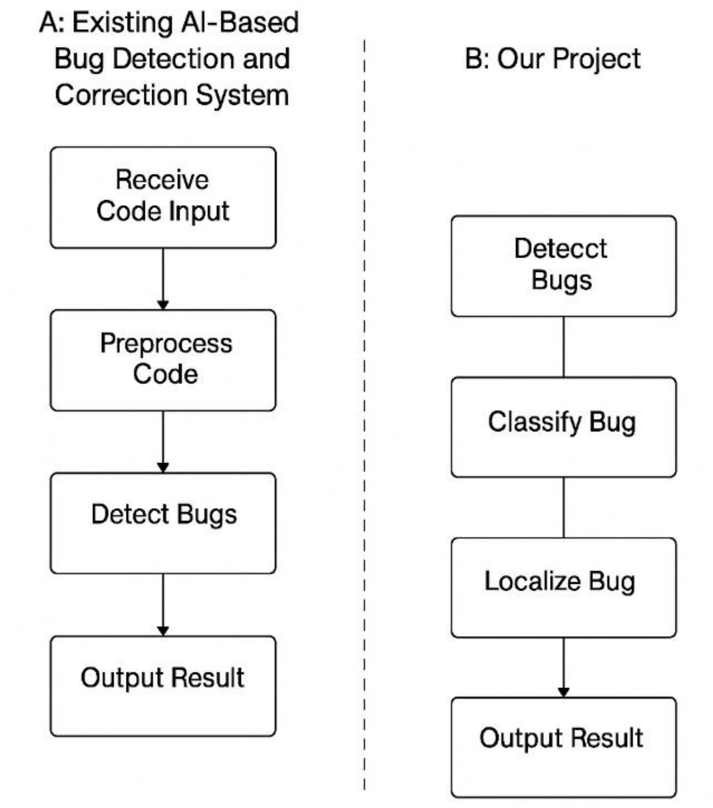


Fig. 1.3: Comparison Between Existing Systems and the Proposed System

In recent years, advances in machine learning (ML) and natural language processing (NLP) have brought intelligent code understanding into the spotlight. Transformer-based models such as T5, CodeBERT, CodeT5, and CodeLlama have shown strong potential for learning patterns in source code and supporting tasks like bug detection, classification, and automatic repair. A few research systems—such as DeepFix, BugsLab, and CODIT—have explored this space, but most focus on one part of the debugging pipeline (e.g., only detection or only fixing) and lack full integration. Currently, there is no widely available end-to-end solution that can detect bugs, classify their type, locate the faulty lines, and provide meaningful fixes—all in one system and optimized for real-world Python code.

This project proposes a complete AI-powered debugging system that does all four tasks: detection, classification, localization, and automated fixing. It uses a fine-tuned CodeLlama model and a custom dataset of buggy and clean code examples to deliver accurate and context-aware results. A detailed discussion of existing approaches, their limitations, and how our solution improves upon them is provided in Chapter 2: Literature Survey.

1.7 Organization of the Report

This report is divided into five chapters:

- Chapter 1: Introduction – Introduces the project, its motivation, objectives, and scope.
- Chapter 2: Literature Review – Surveys existing work and related technologies in AI-based bug detection and code modeling.
- Chapter 3: Development Methodology – Describes the system architecture, dataset preparation, model training, and overall workflow.
- Chapter 4: Technology Stack – Describes all the technologies such as the programming language, Machine learning models and framework, development environment and CPU/GPU usage.

- Chapter 5: System Architecture and Workflow – Describes the overall architecture and methodology of our project including the workflow and the architecture which explains the step by step procedure of the project.
- Chapter 6: Implementation Details and Experimental Results – Covers practical development, sample results, evaluation metrics, and performance analysis.

Chapter 2

LITERATURE REVIEW

The fields of automated software debugging and machine learning for code have seen rapid advancements over the last decade. With the rise of large language models (LLMs) and transformer-based architectures, it is now possible to develop systems that not only understand code but also detect and repair bugs. Traditionally, software debugging has been a manual and time-consuming process involving developers reviewing and testing code. However, with the increase in software complexity and the shift towards faster deployment cycles (e.g., CI/CD pipelines), automated approaches to bug detection and fixing are becoming essential. This chapter presents a comprehensive review of the existing research and technologies related to our project. It highlights the evolution of bug detection methods, the role of machine learning and deep learning in code analysis, and the recent developments in language models like CodeLlama**.** The chapter concludes by identifying the gaps in current solutions that our project aims to address.

2.1 Traditional Bug Detection Techniques

Before artificial intelligence (AI) became prominent in software development, bug detection primarily relied on traditional, manual methods. One common approach was manual code reviews, where developers would inspect each other's code to identify errors such as logical mistakes or syntax issues. While this method fostered collaboration and improved code quality, it was often slow, subjective, and error-prone. Static code analyzers like Pylint, SonarQube, and flake8 offered rule-based scanning of source code to catch syntax violations and bad practices without executing the code. Although these tools could flag certain types of problems early, they lacked the intelligence to detect deeper logical errors.

Unit and integration testing frameworks such as unittest, PyTest, and Nose enabled developers to write test cases to check whether specific functions behaved as expected.

However, these tests required manual effort to create and maintain. Another widely used method involved debugging tools like PDB and debuggers integrated within IDEs, which allowed step-by-step code execution to locate bugs. Yet, these methods heavily relied on human reasoning and were inefficient for large codebases or rapid deployment environments. As software complexity has grown, and with the adoption of Continuous Integration and Continuous Deployment (CI/CD) pipelines, the need for more scalable, intelligent, and automated bug detection methods has become evident.

2.2 Machine Learning in Code Analysis

The incorporation of machine learning (ML) into programming has led to a field known as "ML for code" or "code intelligence." ML enables systems to learn patterns from vast amounts of code examples, making it useful for various tasks. One key application is *bug prediction, where models forecast which sections of code are likely to be defective. Another use is code classification, such as identifying whether a function is buggy or clean. ML also powers code suggestion and completion, enhancing developer productivity by autocompleting function names or generating relevant lines of code. More advanced systems can even perform automated repair, generating patches for buggy code based on learned fixes from large datasets.

Tools like DeepBugs leverage embeddings of code tokens to spot anomalies, while architectures such as Code2Vec and Code2Seq create vector representations of code snippets for tasks like classification and summarization. These techniques are trained on large-scale datasets such as CodeSearchNet, BigCode, and QuixBugs, allowing them to generalize across diverse codebases and styles. Compared to traditional rule-based systems, ML approaches have demonstrated superior adaptability, especially when encountering varied programming logic and syntax.

2.3 Deep Learning and Transformer Models

Deep learning—especially through transformer architectures—has revolutionized how machines interpret and analyze code. The transformer model, introduced by Vaswani et al.

in the seminal paper “Attention Is All You Need” in 2017, enables attention-based learning across input sequences. This mechanism allows the model to capture complex dependencies within code, both over short and long ranges, more effectively than earlier models like RNNs and LSTMs. A number of transformer-based models have emerged, each with a specific focus. CodeBERT, for example, is trained on paired code and natural language to support tasks like classification, summarization, and retrieval. GraphCodeBERT enhances this by incorporating abstract syntax tree (AST) information, giving the model a deeper structural understanding of code. CodeT5, modeled after Google’s T5 architecture, uses an encoder-decoder setup for tasks like code translation, summarization, and fixing. PolyCoder is designed to work well with low-level programming languages and focuses on performance. Codex, developed by OpenAI, is fine-tuned on vast amounts of public code and powers tools like GitHub Copilot, providing real-time code suggestions. The self-attention mechanism of transformers allows these models to understand the full context surrounding a bug, which is crucial for identifying and resolving complex code issues.

2.4 Bug Detection and Repair Using AI

AI-driven approaches to bug detection and fixing are transforming software maintenance into a data-intensive process. Early systems like DeepFix (2016) demonstrated the potential by automatically correcting C language syntax errors using deep learning models trained on student code submissions. In 2019, Tufano et al. introduced transformer-based models trained on real-world bug-fix commit data, focusing on Java code. More recent work includes CoCoNut (2021), which uses context-aware encoders to generate code patches with a deeper understanding of the surrounding code.

BugLab, developed by Meta in 2023, employed a self-supervised learning approach by synthetically injecting bugs into code and training a model to fix them, thereby improving robustness. RETR, another innovative system, combines retrieval-based and generation-based methods: it searches for similar historical buggy code and adapts the corresponding fix to the new context. Despite their promise, many of these models are limited in scope—they often work only on specific programming languages and are not yet adept at handling complex, multi-line bugs or intricate logic errors. These limitations reveal key gaps that current AI models need to overcome for broader and more reliable adoption.

2.5 About CodeLlama

Code Llama is an open-source large language model introduced by Meta, specifically designed for programming and software development applications. Built upon the Llama 2 foundation, Code Llama has been fine-tuned on extensive code datasets across multiple programming languages including Python, C++, and JavaScript. It is available in several parameter sizes—7B, 13B, 34B, and 70B—allowing users to balance computational efficiency with performance needs. There are also three major variants tailored for different use cases: a base model for general code tasks, a Python-specialized model for high performance in Python-centric environments, and an instruct-tuned version optimized to follow natural language instructions.

Code Llama is capable of handling tasks such as code generation, auto-completion, debugging, and infilling. One of its standout features is its ability to manage long contexts—up to 100,000 tokens—making it ideal for analyzing large codebases. Furthermore, Code Llama is freely available under Meta's community license, enabling open research and development. Its capabilities make it an invaluable asset for developers, educators, and researchers seeking to enhance productivity and accuracy in code-related tasks using AI.

2.6 Uniqueness of the Proposed System

While many existing platforms such as Google Colab, PyCharm, and other modern IDEs provide features like error highlighting and tracebacks, they are limited to either detecting errors or offering basic debugging support. These tools typically handle only syntax and runtime errors, displaying the problem without suggesting or applying any fixes. Moreover, they lack contextual understanding of code logic, making them ineffective in detecting deeper semantic or logical issues. Traditional rule-based linters and static analyzers rely on predefined patterns and heuristics, which makes them brittle and unable to generalize across diverse coding styles and problem contexts.

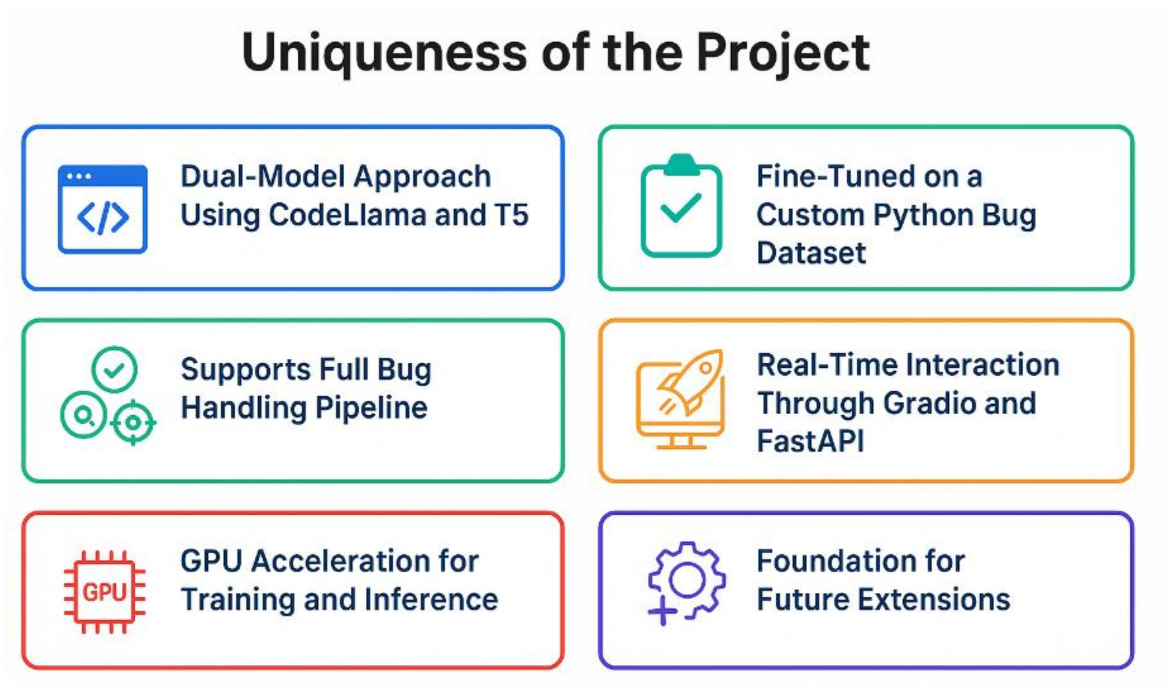

Uniqueness of the Project
Dual-Model Approach Using CodeLlama and T5
Fine-Tuned on a Custom Python Bug Dataset
Supports Full Bug Handling Pipeline
Real-Time Interaction Through Gradio and FastAPI
GPU Acceleration for Training and Inference
Foundation for Future Extensions

Fig. 2.1: Infographic Showing the Uniqueness of the Project

Even machine learning-based systems like DeepFix or CodeBERT tend to focus on a single aspect of debugging, such as code repair or classification, but do not offer an integrated, end-to-end solution. In contrast, our proposed system provides a complete pipeline that automates the full debugging process—including bug detection, classification by type (syntax, logical, runtime, semantic), precise localization of the buggy line(s), and finally, automated code correction. What sets our system apart is its fine-tuned CodeLlama-Instruct model, trained on real-world buggy/fixed Python code pairs, which enables it to generate accurate and context-aware fixes. Additionally, the system offers a user-friendly command-line interface that guides users step-by-step, making the debugging process more intuitive and significantly reducing time complexity.

Unlike existing tools that merely highlight the error, our system interprets the bug, understands its category, pinpoints its location, and produces a corrected version—all in a single, seamless flow. This holistic approach not only saves time but also enhances productivity, especially in scenarios where manual debugging would be tedious or error-prone. The combination of modular architecture, deep learning integration, and real-time interactivity makes our system uniquely effective and scalable for a wide range of users.

Chapter 3

DEVELOPMENT METHODOLOGY

Overview

In this chapter, we present the step-by-step methodology followed to design, build, and evaluate an intelligent system capable of detecting and fixing bugs in Python code. The system is built upon the fine-tuned version of the CodeLlama language model, adapted specifically for handling code. Our approach combines elements of dataset engineering, bug simulation, deep learning fine-tuning, and performance evaluation to create a fully automated pipeline that mirrors how a human might detect and correct programming errors. The methodology begins with the preparation of a suitable dataset that includes both clean and buggy Python code.

This is followed by synthetic bug injection to generate diverse examples for training the model. The core model, CodeLlama, is then fine-tuned on these data samples, and additional modules are added for bug classification, localization, and code correction. Each stage has been carefully designed to support the goals of high accuracy, generalization across bug types, and practical utility for Python developers.

3.1 System Architecture Overview

The architecture of our automated bug detection and repair system is designed as an end-to-end pipeline that facilitates a seamless flow from data ingestion to code correction. The system is structured to mirror a real-world debugging process, beginning with the intake of raw Python code and culminating in the delivery of a corrected version of that code, along with an explanation of the detected bug's nature and location. The core innovation lies in the integration of several automated modules, each responsible for a specific phase: dataset preparation, bug injection and annotation, model fine-tuning, bug detection and classification, localization, and automated code repair. At the heart of the system is the CodeLlama model, a large language model (LLM) fine-tuned specifically for Python.

The architecture leverages this model's ability to comprehend and manipulate programming constructs in a way that mimics human-level code understanding. Unlike traditional tools that require static datasets and predefined rules, our system dynamically simulates code errors, fine-tunes models on-the-fly, and evolves with new data. The input to the system is a piece of Python code—either complete or partial—which passes through a bug detection module. If a bug is found, the system classifies the bug type, pinpoints its exact location in the code, and forwards it to the fixing module.

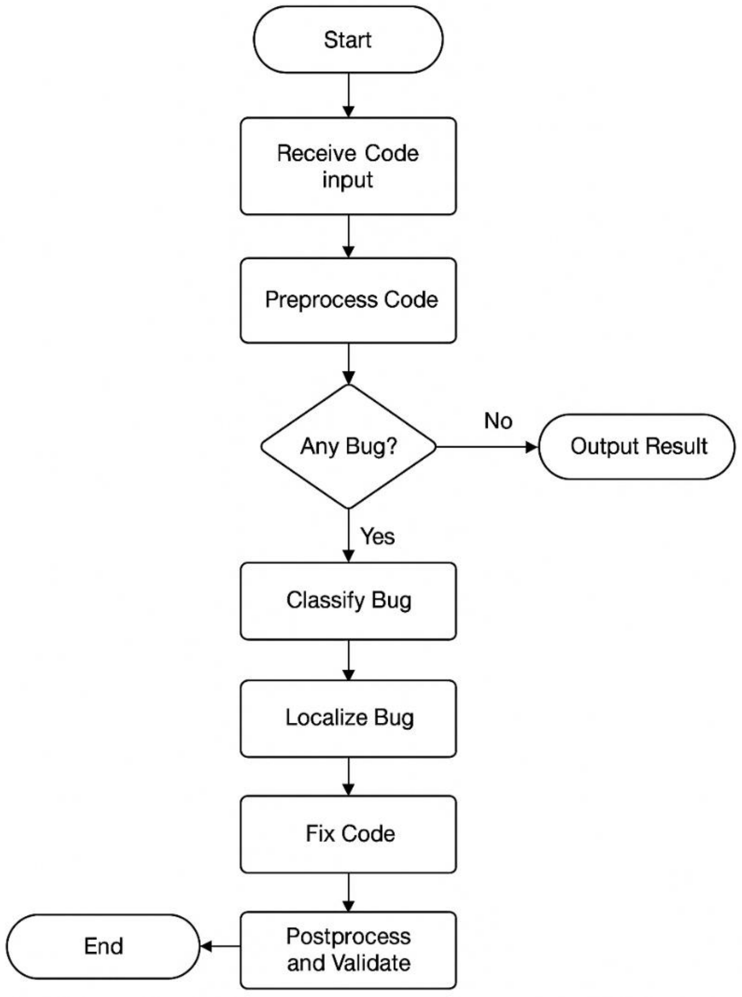


Fig. 3.1: System Architecture and Workflow

Each component is designed to work in isolation as well as in collaboration, ensuring modularity and scalability. For example, the bug injection framework is not just a data augmentation tool but also acts as a benchmark generator for new models. Likewise, the localization module’s outputs are critical for improving user feedback and trust.

A particularly innovative aspect of our system is its self-supervised learning capability, which allows it to train and adapt without heavy reliance on manually labeled datasets. This architectural flexibility ensures that our system can be deployed in various development environments—from IDE plugins to CI/CD pipelines—and can support future improvements such as real-time code monitoring or multi-language support.

3.2 Dataset Preparation

In any machine learning project—especially one that deals with the complexities of source code—the quality and structure of the dataset directly impact the system’s performance. For our AI-based Python bug detection and fixing tool, we needed a dataset that reflects real-world code variations, includes a wide range of bug types, and provides enough examples for the model to learn meaningful patterns. Since no single dataset fulfilled all these requirements, we chose to build our own dataset from scratch, using multiple sources, structured design, and verification methods.

We began by collecting clean, functional Python code samples from several trusted educational and professional platforms such as W3Schools, GeeksforGeeks, ChatGPT-generated content, and solutions published in IEEE research papers. These platforms offer diverse and high-quality Python functions used in real-world applications, problem-solving platforms (like LeetCode), and academic research. This gave us a mix of basic programming structures, algorithmic logic, and real-world implementation patterns. The collected functions included tasks such as sorting, recursion, file operations, mathematical calculations, string manipulations, and more.

After gathering raw code, we initiated a preprocessing phase where we cleaned each function to ensure uniformity and avoid any noise during model training. This involved removing comments, docstrings, print/debugging statements, and redundant whitespaces. We also ensured consistent indentation, as Python is indentation-sensitive, and incorrect spacing might mislead the model. For parsing and validating structure, we used Python's

Abstract Syntax Tree (AST) module, which allowed us to understand the function hierarchy and extract token-level information systematically.

Next, we developed a custom bug injection module that introduced controlled errors into clean code. This was essential for training the model to not just recognize what clean code looks like, but to also understand how bugs manifest. The injected bugs were realistic and diverse, covering:

- Syntax Errors (e.g., missing colons, unmatched parentheses)
- Logical Errors (e.g., incorrect conditionals, misplaced return statements)
- Runtime Errors (e.g., type mismatches, undefined variables)
- Semantic Errors (e.g., wrong use of variables, incorrect algorithms)

For each buggy version, we created a corresponding fixed version, resulting in paired data consisting of:

- buggy_code: the mutated code with injected error(s)
- fixed_code: the original correct version
- bug_type: classification of the bug (e.g., SyntaxError, LogicalError)
- bug_line: the specific line(s) where the error occurred
- severity_level: optional labels indicating the complexity or impact of the bug (e.g., minor, critical)

All data samples were stored in structured.jsonl (JSON Lines) format, which allowed us to process the data efficiently in streaming pipelines during training and testing. To ensure the reliability of this custom dataset, we ran validation tests on Google Colab, using smaller subsets of data and executing the buggy and fixed code to verify correctness. We also used the CodeLlama-Instruct model in early experiments to test whether it could accurately identify and correct these bugs, which served as both a functional test and a form of soft validation for dataset quality.

The entire dataset was then split into three parts using a 70:15:15 train-validation-test split, ensuring balanced representation of bug types and difficulty levels in each subset.

This careful partitioning allowed us to track model performance across training phases and prevented overfitting by avoiding data leakage. What makes our dataset preparation unique is the combination of synthetic and real-world code, customized annotation, and compatibility with large language models like CodeLlama. The modular design of the data structure also ensures that we can extend it in the future to support multi-language debugging, additional bug types, or even context-aware debugging (e.g., bugs related to code dependencies or APIs).

3.3 Bug Injection and Annotation

In machine learning-based bug detection systems, one of the most significant challenges is the scarcity of labeled data that pairs buggy code with its corrected version. Real-world bug-fix datasets are limited, especially for Python, and collecting such data manually is both time-consuming and error-prone. To address this gap, we developed a sophisticated bug injection framework designed to artificially introduce realistic bugs into clean code samples. The process starts with our curated dataset of syntactically valid Python code. From this baseline, we systematically introduce a wide variety of bugs that mirror real-world programming mistakes.

These include syntax errors like missing colons, unmatched parentheses, and incorrect indentation; logical errors such as using <= instead of <, or swapping operands in conditional statements; and runtime errors like undefined variables, incorrect function calls, and type mismatches. Each bug is injected with careful consideration to ensure that the resulting code remains realistic and does not become trivially broken or overly synthetic. Alongside each injected bug, we retain the original code snippet as the target fix, effectively creating a labeled pair of buggy and corrected code. To make the dataset even more valuable for downstream tasks like classification and localization, each injected bug is annotated with metadata. This includes the type of bug (e.g., SyntaxError, LogicalError), the specific line number where the bug was introduced, and a brief textual description of the error. These annotations serve multiple purposes: they guide the model during supervised learning, enable fine-grained evaluation of its capabilities, and allow for explainable outputs during inference.

By automating the bug injection and annotation process, we were able to generate thousands of high-quality training examples without manual intervention. This methodology not only made our dataset richer and more diverse but also ensured that our model could learn from a broad spectrum of realistic code issues. In doing so, we significantly lowered the barrier to developing effective AI-driven debugging tools.

3.4 Model Selection and Fine-Tuning

Selecting the appropriate machine learning model is a critical decision in any AI project, particularly when dealing with complex tasks like code understanding and bug repair. For our system, we selected the CodeLlama model, specifically its Python-optimized and instruction-tuned variants. CodeLlama, developed by Meta, is a large language model based on the Llama 2 architecture and trained specifically on code across multiple programming languages, with a deep understanding of Python syntax, semantics, and idiomatic patterns.

What makes CodeLlama especially suitable for our application is its ability to follow human-readable instructions and generate structured output, such as fixed code snippets. To adapt the model more closely to our domain-specific task—debugging Python code—we employed a process of fine-tuning. Fine-tuning allows a pre-trained model to adjust its parameters based on new, task-specific data—in our case, pairs of buggy and fixed code samples. We used the HuggingFace Transformers framework for training and evaluation, and implemented LoRA (Low-Rank Adaptation), a fine-tuning method that introduces small, trainable layers into the model's architecture.

LoRA makes the fine-tuning process significantly more efficient by reducing the number of parameters that need updating, which is particularly useful when dealing with large models like CodeLlama. The optimizer used during training was AdamW, which combines adaptive learning rates with weight decay to prevent overfitting. We evaluated the model's performance using several metrics, including accuracy on the validation set and BLEU score for text generation similarity.

During training, the model learned to map buggy code to its corrected version by recognizing patterns associated with common programming mistakes. The instruction-tuned version was particularly useful in helping the model interpret prompts like "Fix this function" and respond appropriately. By the end of fine-tuning, the model was not only capable of correcting a wide variety of bugs but also generalized well to new, unseen errors, demonstrating the effectiveness of our model selection and training methodology.

3.5 Bug Detection and Classification

After fine-tuning, our model gained the ability to differentiate between clean and buggy code. However, for the system to be more informative and developer-friendly, we extended its functionality to include bug classification. This means that instead of simply stating that an error exists, the system can also explain what kind of error it is—such as a SyntaxError, LogicalError, NameError, or TypeError. To implement this, we added a classification head on top of the transformer's final hidden layer outputs. This head was trained in parallel with the main sequence-to-sequence bug-fixing model, using the same dataset but with emphasis on the bug type annotations.

These annotations were generated during the bug injection phase and helped the model learn fine-grained distinctions between different bug categories. By learning from these distinctions, the classification head enables the model to provide contextual explanations for its fixes, thereby improving transparency and user trust. During evaluation, we measured the classification accuracy independently from bug detection and code repair metrics, allowing us to fine-tune the model's understanding of error categories.

This classification capability is especially useful in educational tools and integrated development environments (IDEs), where developers often benefit from knowing not just that their code is broken, but why. It also opens up possibilities for future features such as error-specific fix suggestions, prioritization of bugs based on severity, and dynamic learning where the model adapts to error patterns specific to a user or project.

3.6 Bug Localization

An essential component of any debugging system is its ability to localize the bug—that is, to pinpoint the exact line in the code where the error occurs. Without precise localization, even a correctly classified or repaired bug might not be helpful in practical scenarios. For our system, we emphasized line-level bug localization to enhance usability, especially when integrating the tool into real-time development environments or CI/CD pipelines. To accomplish this, we trained the model to predict the line number most likely to contain the bug. This was done using the attention mechanisms inherent in transformer models, which naturally assign higher weights to important tokens during processing.

By analyzing the attention scores across tokens and mapping them to their respective lines, we were able to approximate which lines the model "focused" on when identifying bugs. In parallel, we experimented with token-level classification, where each token in the code was assigned a probability of being part of a buggy construct. These token-level probabilities were then aggregated to the line level, providing a ranked list of suspicious lines.

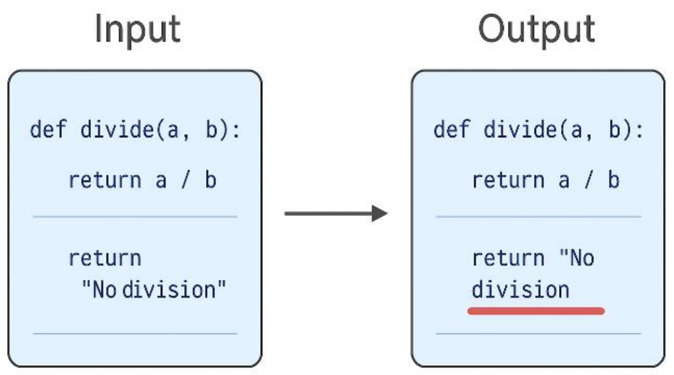


Fig. 3.2: Bug Localization Workflow – Line-Level Error Identification

During training, the ground truth line annotations created during the bug injection phase served as the target for the localization model. The model was trained to minimize cross-entropy loss over the predicted line indices, and its performance was validated using metrics like top-1 and top-3 localization accuracy. Accurate localization not only improves the debugging experience but also allows partial code correction, where only the buggy part of the code is edited, preserving the rest.

This is particularly useful in collaborative environments where minimal disruption to working code is desired. In summary, the localization component transforms the system from a generic code editor into an intelligent assistant capable of guiding developers directly to the source of their problems.

3.7 Automated Bug Fixing

The final and most impactful stage of our system is the automated bug fixing module, which transforms detected and localized errors into corrected code. This component is the culmination of all prior modules—after identifying that a bug exists, determining its type, and locating it in the code, the system proceeds to generate a correct and functional version of the code snippet. We leveraged the instruction-tuned variant of the CodeLlama model for this task, as it is designed to follow human-like prompts and return structured outputs. The core idea here is to prompt the model with both the task and the buggy code.

A typical input might be: "Fix the following Python function and return the corrected version," followed by the buggy code block. The model, having been fine-tuned on thousands of buggy-correct code pairs, generates a syntactically valid and semantically accurate fixed version of the input. This is not simply a regurgitation of memorized fixes but rather an intelligent transformation based on understanding the context and structure of the code.

To ensure that the fixes are valid, we implemented a dual-layer validation mechanism. First, we compare the model's output against the ground-truth fixes from our dataset. This gives us an accuracy score and helps track how well the model reproduces known solutions. Second, we execute the fixed code and check if it runs without raising exceptions and whether it passes predefined test cases. These tests are either unit tests derived from the original problem (in the case of educational or challenge-based datasets) or custom test inputs designed to validate output correctness. Additionally, we employed beam search during generation to allow the model to produce multiple candidate solutions.

The beam search strategy retains the top ‘n’ most likely outputs based on the model’s confidence and then ranks them based on execution success or similarity to the expected solution. This increases the chances of finding a successful fix, especially for more complex bugs where a single deterministic output might not suffice.

While challenges remain for deeper logical or multi-function bugs, the model often manages to generate plausible and testable corrections. This module ultimately embodies the vision of AI-assisted programming: an intelligent system that not only points out issues but actively contributes to writing cleaner, functional, and bug-free code.

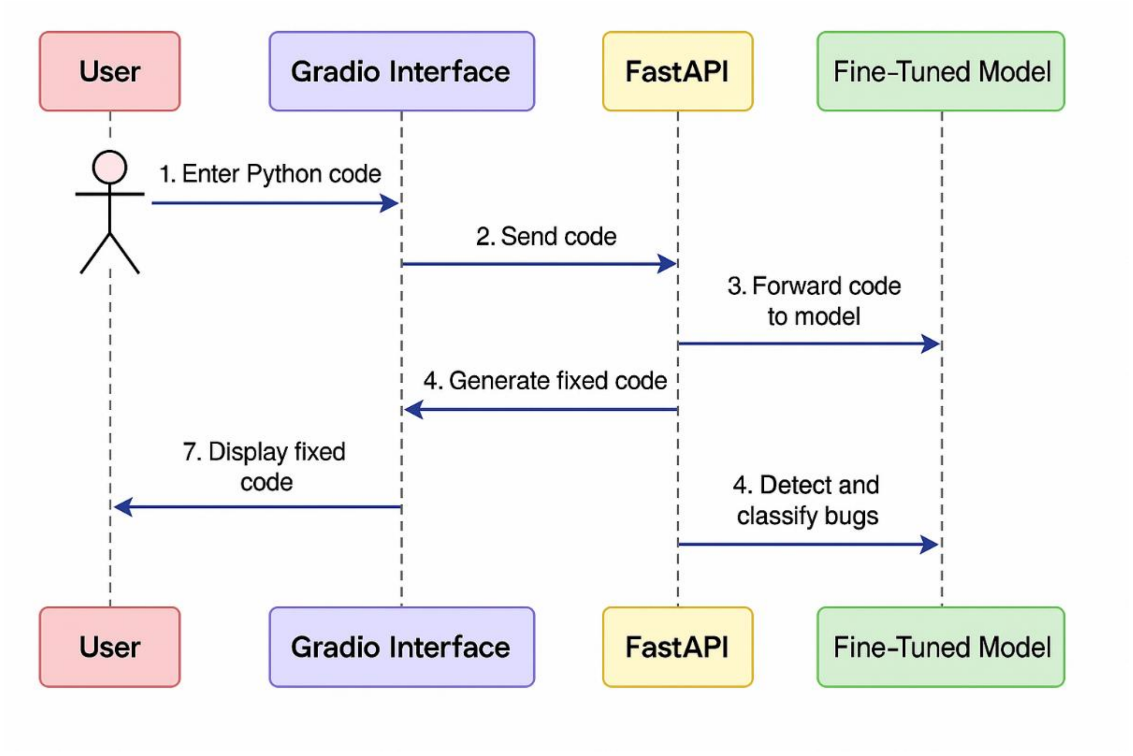


Fig. 3.3: Color-Coded Sequence Diagram of Bug Detection System

The system also supports “explanation generation” as a byproduct of the instruction-tuned model’s architecture. With minor modifications, the model can be asked to not only provide the corrected code but also briefly describe what was fixed. For example, it might append a comment like “# Fixed a missing colon at the end of the if statement.”

This feature is particularly useful for educational purposes, where users benefit not only from having their errors corrected but also from understanding the rationale behind those corrections. In practice, the automated bug fixing module has shown robust performance, particularly with single-line and syntax-level bugs.

3.8 Evaluation Metrics

Evaluating an AI-powered debugging system requires a multi-faceted approach, as different components contribute to the overall user experience and effectiveness. To this end, we established a comprehensive evaluation framework using both quantitative and qualitative metrics. The first set of metrics pertains to bug detection. We calculated detection accuracy by measuring the percentage of buggy code samples that were correctly identified as containing an error.

This binary classification task (buggy vs. clean) is fundamental because a failure to detect a bug renders the downstream modules irrelevant. Next, we focused on classification accuracy, which gauges how accurately the model assigns the correct bug type (e.g., SyntaxError, LogicalError). This was evaluated using standard multi-class classification metrics such as precision, recall, and F1 score for each class.

For the localization component, we introduced line-level accuracy metrics. Top-1 accuracy measures whether the model's most confident prediction matches the actual line of the bug, while Top-3 accuracy allows for partial credit if the correct line appears among the top three predictions. These metrics are especially important in practical usage scenarios, where even an approximate indicate on of the bug location can guide developers toward a solution. Moving forward, the fix success rate became the core performance indicator for the automated repair module. This was computed as the percentage of buggy code samples for which the model-generated fix resolved the error entirely, as verified by either exact match to the ground-truth fixed code or by passing predefined test cases during runtime execution.

Additionally, we used the BLEU (Bilingual Evaluation Understudy) score to compare the similarity between the predicted fixes and the actual corrected code. While

BLEU is traditionally used in natural language translation tasks, it serves a valuable role here in assessing whether the generated fix structurally resembles the expected solution. However, BLEU does not account for functional correctness, so we supplemented it with execution-based validation. In this method, we ran the fixed code and evaluated whether it executed without errors and produced the correct output for a set of test inputs. This helps uncover cases where the fix may look syntactically correct but still fails to produce the desired behavior.

Lastly, we included error analysis as part of our qualitative evaluation. This involved manually reviewing a subset of failed fixes to understand common failure modes—such as off-by-one errors, incorrect variable names, or misunderstanding of deeper logic. This manual analysis not only highlighted the model's limitations but also suggested improvements for future training, such as incorporating more diverse logical bugs or edge-case scenarios. Altogether, the evaluation metrics provide a rich picture of the system's strengths and weaknesses. High detection and classification scores indicate reliable identification capabilities, while strong localization and fix success rates validate the system's practical utility. By quantifying performance at each stage, we ensured that our model development was guided by measurable progress and that the final system delivered real-world value to developers and learners alike.

Chapter 4

TECHNOLOGY STACK

In any software development project—especially those involving artificial intelligence and machine learning—the choice of technology stack plays a crucial role in determining the success, performance, scalability, and maintainability of the system. This chapter presents a comprehensive overview of the technology stack used to implement our AI-based Python bug detection and correction system. Each tool and framework has been carefully selected to meet the computational, architectural, and user-interface needs of the project. The system leverages advanced natural language processing (NLP) models such as CodeLlama and T5, which are built on transformer-based architectures.

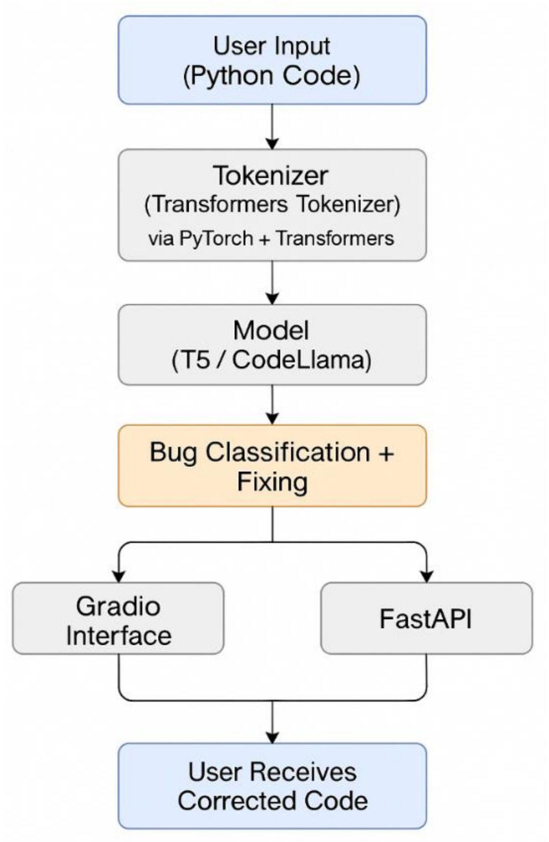

User Input (Python Code)
Tokenizer (Transformers Tokenizer) via PyTorch + Transformers
Model (T5 / CodeLlama)
Bug Classification + Fixing
Gradio Interface
FastAPI
User Receives Corrected Code

Fig. 4.1: Workflow of Model-Based Bug Detection and Classification

Figure 4.1 provides a high-level overview of the complete bug detection and fixing workflow, starting from user code input to final output through interactive interfaces like Gradio and FastAPI.

As such, we employed modern deep learning frameworks like PyTorch and Hugging Face Transformers, supported by CUDA for GPU acceleration. On the deployment and user interaction front, web technologies like FastAPI and Gradio were integrated to build APIs and intuitive user interfaces. The development and runtime environments included platforms like PyCharm for local development and Google Colab Pro for GPU-enabled cloud training. The entire codebase was managed using Git and GitHub, which provided robust version control, collaboration, and code sharing.

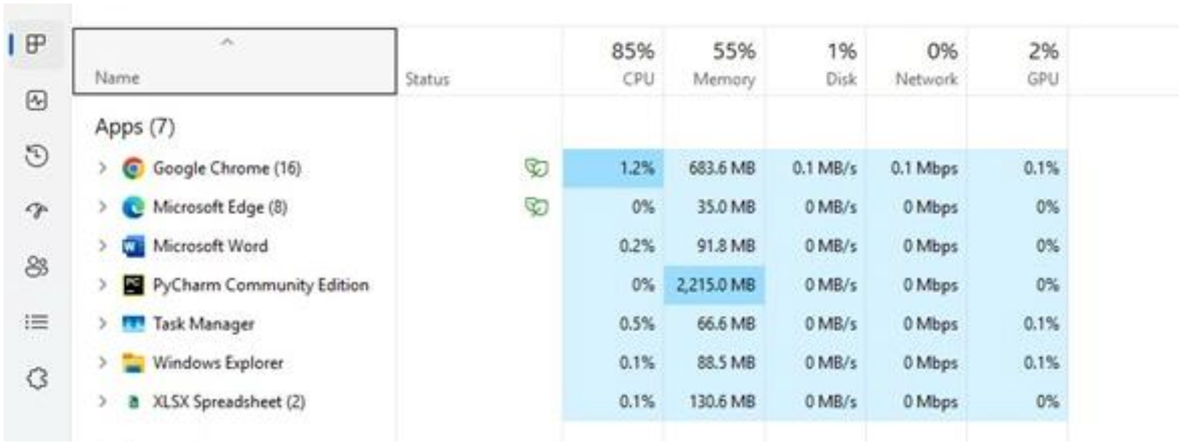
4.1 Programming Language

Python serves as the primary programming language for the entire development lifecycle of the automated bug detection and correction system. It is chosen not only for its simplicity and readability but also for its extensive ecosystem of libraries and tools that are specifically tailored for machine learning, data processing, and system integration. Python is used in every stage of the project—from initial data preprocessing to advanced model training, evaluation, and deployment.

During data preprocessing, Python libraries such as Pandas and NumPy are used to clean, organize, and transform code datasets (buggy and corrected code pairs) into a format suitable for machine learning models. For model development and fine-tuning, Python interfaces seamlessly with deep learning frameworks like PyTorch and Hugging Face Transformers, allowing developers to implement, customize, and train large language models such as CodeLlama and T5. In the evaluation stage, Python's Scikit-learn library helps calculate key performance metrics like accuracy, precision, recall, and F1-score. Moreover, Python's compatibility with GPU acceleration frameworks like CUDA makes it ideal for high-performance tasks, enabling faster training and inference times. It also supports user interface development through libraries such as FastAPI, Gradio, and Streamlit, which help create interactive web applications for real-time code analysis and correction. Overall, Python's versatility, community support, and integration capabilities make it the optimal language for building a scalable and intelligent bug detection system.

4.2 CPU vs. GPU Utilization

In the initial phases of development, the bug detection system was executed solely on a CPU-based environment. This led to 100% CPU utilization, which significantly slowed down the process of model training and inference. Since transformer-based models like T5 and CodeLlama involve large-scale matrix operations and complex computations, the CPU struggled to keep up with the intensive workload. This high usage caused latency issues, longer training times, and limited the system's ability to process large datasets or fine-tune bigger models effectively. To address this performance bottleneck, the system was migrated to a GPU-accelerated environment. By leveraging NVIDIA GPUs along with the CUDA (Compute Unified Device Architecture) platform, the project experienced a substantial improvement in performance.



Name	Status	85% CPU	55% Memory	1% Disk	0% Network	2% GPU
Apps (7)						
Google Chrome (16)		1.2%	683.6 MB	0.1 MB/s	0.1 Mbps	0.1%
Microsoft Edge (8)		0%	35.0 MB	0 MB/s	0 Mbps	0%
Microsoft Word		0.2%	91.8 MB	0 MB/s	0 Mbps	0%
PyCharm Community Edition		0%	2,215.0 MB	0 MB/s	0 Mbps	0%
Task Manager		0.5%	66.6 MB	0 MB/s	0 Mbps	0.1%
Windows Explorer		0.1%	88.5 MB	0 MB/s	0 Mbps	0.1%
XLSX Spreadsheet (2)		0.1%	130.6 MB	0 MB/s	0 Mbps	0%

Fig. 4.2: CPU & GPU Utilization

Model training and inference became up to 10 times faster than on CPU. Surprisingly, even with such performance gains, GPU utilization remained relatively low—averaging around 2%, which indicates efficient use of resources and potential for further scaling. This shift to GPU not only reduced training time but also enabled real-time predictions and allowed the use of larger batch sizes during model training, which is crucial for improving model accuracy and stability.

4.3 Language Models: T5 and CodeLlama

The system incorporates advanced transformer-based models to perform automated bug detection and correction, with T5 (Text-to-Text Transfer Transformer) and CodeLlama playing central roles. Initially, T5 was used in exploratory phases to conceptualize bug correction as a text-to-text generation task, wherein buggy code is transformed into its corrected version. However, the core of the system relies on CodeLlama**,** a specialized large language model designed specifically for understanding and generating source code. CodeLlama was fine-tuned on a curated dataset consisting of Python buggy and fixed code pairs. This fine-tuning process improved the model's ability to identify coding errors and generate_accurate_fixes.

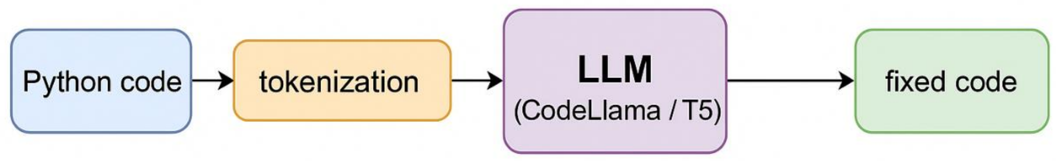


Fig. 4.3: Architecture of CodeLlama

4.4 Machine Learning Frameworks

The machine learning backbone of the system is built using a combination of state-of-the-art frameworks and libraries that facilitate efficient model training, fine-tuning, and deployment. At the core is the Hugging Face Transformers library, which provides an intuitive and powerful interface for accessing and customizing large pre-trained models like T5 and CodeLlama. This library plays a crucial role in ensuring that the models can be integrated into the debugging pipeline with minimal overhead while offering extensive support for tokenization, model configuration, and training workflows. The use of this high-level API significantly reduces the complexity of handling massive language models and allows for seamless execution across both CPUs and GPUs.

Underlying the Transformers library is PyTorch, a widely adopted deep learning framework known for its dynamic computational graph and ease of use. PyTorch supports efficient GPU acceleration via CUDA and is particularly suited for research-oriented projects due to its flexibility in defining custom model behaviors. All major training operations, including loss computation, backpropagation, optimization, and checkpointing, are implemented using PyTorch functionalities, allowing developers full control over the learning process. Moreover, tokenization—a vital preprocessing step for any transformer model—is handled using Hugging Face’s Tokenizers library. Unlike traditional NLP tokenizers, this library is optimized for programming languages and retains the structural integrity of source code.

Input
data
Tensor
Output

Fig. 4.4: Architecture of CodeLlama for Bug Detection and Fixing

The tokenization scheme ensures that elements such as indentation, symbols, and syntax structures are preserved, thereby improving the model's ability to learn meaningful code representations. Together, these tools provide a powerful infrastructure for training and deploying large transformer models capable of performing complex tasks such as bug detection, classification, and automatic code repair.

4.5 Web Frameworks

To provide a user-centric and accessible experience, the system integrates two modern web frameworks—FastAPI and Gradio—which enable both backend API services and frontend interaction interfaces. FastAPI serves as the backbone for the system’s web server, offering a high-performance, asynchronous web framework based on Python. Its ease of use and support for automatic OpenAPI documentation makes it ideal for deploying machine learning models as RESTful APIs. Through FastAPI, developers can expose endpoints that accept raw Python code as input, pass the input to the bug detection model, and return predictions in real time.

These endpoints can be consumed by a variety of frontends, including web apps, desktop clients, and mobile interfaces, ensuring that the system can be accessed and integrated across diverse platforms. In parallel, Gradio is employed to build an intuitive graphical user interface (GUI) that makes the system accessible to users without programming experience. With Gradio, users can interact with the bug detection model directly through a web browser. They can paste or type Python code into an input field, press a button, and instantly view the corrected version along with details about the identified bug.

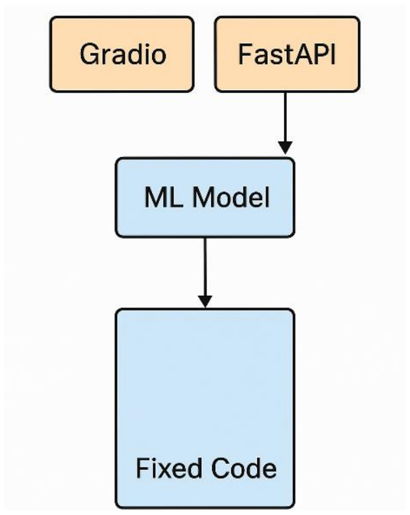


Fig. 4.5: Web Application Stack for Bug Fixing System

The interaction between frontend tools (Gradio and FastAPI) and the ML model is illustrated in figure 4.5, highlighting the overall structure of the web application layer used in the system.

Gradio abstracts away the complexity of frontend development, allowing the research team to focus on model performance and usability rather than UI engineering.

Additionally, it supports features like example caching, output explanation, and live code execution, which enhance the overall utility of the interface. Together, FastAPI and Gradio ensure that the powerful capabilities of the model are not limited to developers but are also accessible to educators, students, and software engineers through a seamless and interactive platform.

4.6 Development Environments

Developing a complex AI system for bug detection and repair requires a robust and flexible development environment that supports both experimentation and production readiness. For local development, PyCharm is the primary Integrated Development Environment (IDE) used by the team. PyCharm provides a comprehensive suite of features including intelligent code completion, real-time error detection, integrated testing tools, version control support, and debugger utilities, all of which streamline the development and debugging process.

This environment is especially useful when building the preprocessing pipelines, annotating datasets, or writing API integration code. On the other hand, large-scale model training and evaluation are conducted on cloud platforms, specifically Google Colab Pro. Colab Pro offers free access to high-end NVIDIA GPUs such as Tesla T4, P100, and even A100 for limited time slots. This is crucial because training transformer-based models on CPU-only setups is prohibitively slow and resource-intensive. Colab allows the team to leverage GPU acceleration without the need for on-premise hardware, making it ideal for academic or budget-conscious projects.

It also facilitates the use of interactive Jupyter Notebooks, which are particularly useful for tracking experiments, visualizing results, and sharing work among collaborators. In addition to cloud storage provided by Colab, local storage solutions are employed to retain important artifacts such as datasets, fine-tuned model checkpoints, performance logs, and visualizations. This dual approach—local development with PyCharm and cloud-based training with Colab—provides the flexibility to quickly prototype new ideas while also executing resource-heavy training jobs effectively. The hybrid setup ensures that developers can iterate rapidly on their models and features without being bottlenecked by hardware constraints.

As shown in figure 4.6, development is carried out in environments like PyCharm and Google Colab, after which the trained models and data are stored locally for later use during inference.

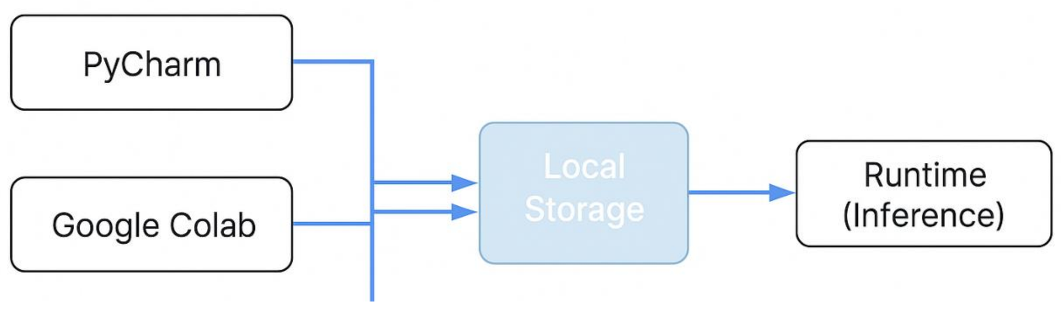


Fig. 4.6: Development Workflow for Model Training and Inference

4.7 Runtime Support: CUDA

To fully exploit the computational demands of deep learning models, particularly transformer-based architectures like CodeLlama, the system’s runtime environment is heavily optimized for GPU acceleration through NVIDIA's CUDA platform. CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface that allows software to leverage the immense parallel processing power of NVIDIA GPUs. Without CUDA, training large models such as CodeLlama or T5 would require days or even weeks on traditional CPUs, making real-time debugging or rapid iteration nearly impossible. CUDA works by offloading matrix multiplications, tensor operations, and backpropagation steps to thousands of GPU cores, enabling high-throughput training and inference.

The system is configured with appropriate CUDA versions and compatible cuDNN (CUDA Deep Neural Network) libraries to ensure full interoperability with PyTorch and Hugging Face Transformers. These configurations are carefully managed to avoid version mismatches that could lead to runtime errors or degraded performance. In Google Colab, CUDA comes pre-installed and automatically linked to the available GPU, which simplifies setup for cloud-based execution. For local machines, CUDA must be installed and configured manually, often involving environment variable settings, driver updates, and library path definitions. The benefits of CUDA extend beyond just training; during inference, CUDA allows the system to respond to user inputs in real time by rapidly processing transformer layers and generating corrected code outputs with minimal latency. Overall, CUDA forms the computational engine of the project, enabling it to scale efficiently, deliver real-time feedback, and meet the high-performance demands of modern AI applications.

4.8 Version Control

Managing a complex AI project involving data pipelines, model training, evaluation scripts, and deployment logic requires rigorous version control. For this purpose, Git is used as the version control system, with GitHub as the remote hosting platform. Git enables the team to track changes in the codebase over time, revert to previous versions, experiment with new features in isolated branches, and collaborate efficiently across different roles. Every significant change, such as updating the model architecture, modifying preprocessing steps, or enhancing the UI, is committed with a descriptive message, ensuring a transparent and documented development history.

Branching strategies are adopted to separate experimental features from the main production code, reducing the risk of introducing bugs during active development. GitHub, being a centralized platform, allows multiple contributors to push changes, create pull requests, and conduct peer reviews before merging updates. It also integrates with CI/CD tools, allowing for automated testing and deployment of changes to staging environments. Additionally, GitHub Issues and Projects are utilized for task tracking, bug reporting, and feature planning, turning the repository into a hub for project management.

Chapter 5

SYSTEM ARCHITECTURE AND WORKFLOW

In any AI-driven software engineering solution, the system architecture plays a critical role in determining how individual modules interact and contribute to the final goal. For our AI-based bug detection and correction system, the architecture is designed to perform a series of tasks: from receiving user code, identifying and classifying potential bugs, locating the precise line(s) responsible, and finally generating an accurate fix. These tasks are performed in a modular and well-orchestrated sequence, ensuring both performance and adaptability. The workflow follows a pipeline model where each stage feeds into the next, yet components remain loosely coupled, allowing for easy upgrades and experimentation.

This chapter offers an in-depth explanation of the entire system architecture. We cover how each component functions—from input handling to code fixing—highlighting the role it plays and how it interacts with the rest of the system. Our architecture is both modular and scalable, supporting future expansion into new programming languages or deeper code analysis. The use of transformer-based models, custom preprocessing pipelines, and post-processing utilities has allowed us to build a system that is not only functional but also efficient and extensible. The following sections walk through each module step-by-step, offering a clear picture of the internal workflow and implementation logic.

5.1 Introduction

The core of any AI-based software system lies in its internal structure—how the different components are connected, how data flows between them, and how each module contributes to the overall functionality. In our project, the goal is to detect, classify, localize, and fix bugs in Python code using a fine-tuned version of the CodeLlama model. To accomplish this, we developed a robust architecture that allows seamless transition from input code to final corrected output. This chapter presents a detailed description of the complete system architecture, its workflow, the role of each module, and how the overall pipeline is designed to support accuracy, flexibility, and scalability.

5.2 Overall System Architecture

The system is organized into a structured pipeline where each stage performs a specific task: starting from receiving the user's Python code, analyzing it for bugs, determining the type and location of bugs, and finally generating a corrected version of the code. The output is then verified for correctness and presented to the user. The major components include the Input Handling Unit, Preprocessing Engine, Bug Detection and Classification Module, Bug Localization Unit, the Code Fixing Module (powered by CodeLlama), and finally, a Postprocessing and Validation Component.

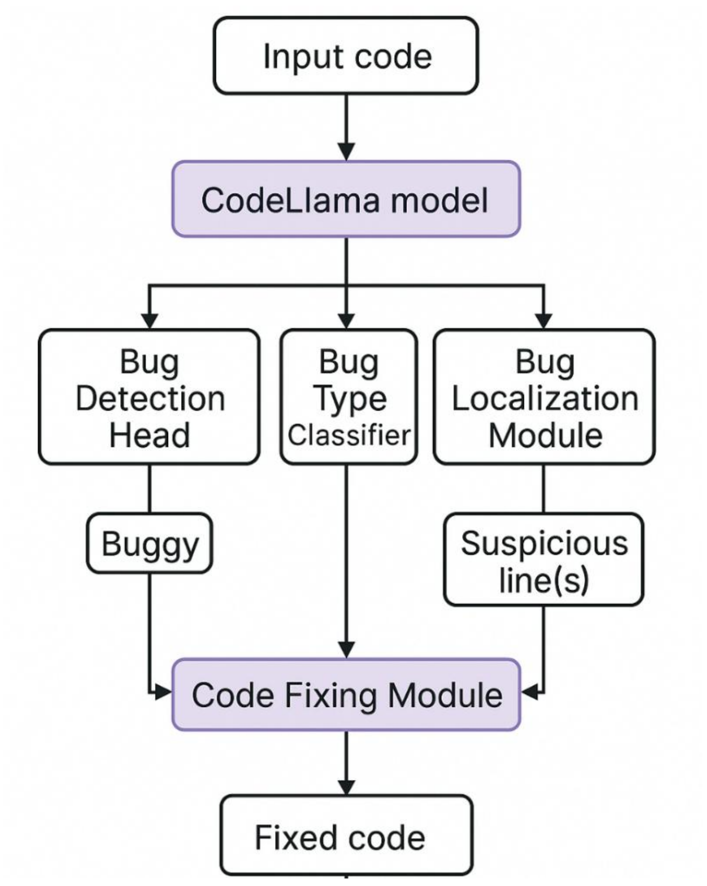


Fig. 5.1: End-to-End System Architecture for Python Code Bug Detection & Fixing

Each module operates independently but communicates with others in a well-defined sequence, making the entire architecture modular and scalable. This pipeline ensures that code transitions from raw input to validated output through a systematic and efficient process. Each stage is also designed to provide explainable insights to users—such as error type and location—improving both trust and usability.

5.3 Input Handling and Preprocessing

The first stage of the pipeline deals with receiving and preparing the input code. Users can input code via different modes—copy-pasting into a command-line interface, uploading a text file, or through an integrated API endpoint. Once received, the Input Handling Unit checks for basic syntactic validity such as matching brackets, correct indentation, and proper encoding. The code then passes into the Preprocessing Engine, which prepares it for machine learning inference. This includes normalization steps such as converting tabs to spaces, removing excessive whitespace, and stripping comments that might mislead the model during tokenization.

Next, the code is segmented at the function level, and each function is assigned structural metadata including line numbers, which are essential for later modules such as bug localization. Tokenization is performed using the tokenizer compatible with the CodeLlama model, converting the cleaned code into a sequence of tokens that preserve the structure and semantics of the original input. This preprocessing ensures consistency, reduces noise, and improves model performance during inference and classification stages.

5.4 Bug Detection and Classification Module

Once preprocessing is complete, the code is passed into the Bug Detection and Classification Module. This is the stage where the system determines whether the code is buggy and, if so, identifies the nature of the bug. This module is built upon a classification head fine-tuned on top of CodeLlama embeddings, which enables it to differentiate between bug-free and buggy functions with high precision.

The classifier categorizes bugs into four major types: syntax errors (e.g., incorrect indentation, missing colons), logical errors (e.g., incorrect loop conditions), runtime errors (e.g., division by zero, use of undefined variables), and semantic errors (e.g., wrong variable usage or wrong API calls). The model produces a binary decision—buggy or not—along with the predicted bug type and a confidence score. This information not only guides downstream modules but also provides valuable feedback to users, making the system more transparent and educational. The classification results significantly influence the strategy used during the code fixing phase, particularly when prioritizing which parts of the code need to be preserved or restructured.

5.5 Bug Localization Module

Bug localization is a critical and technically challenging task that involves identifying the exact line(s) in the code responsible for the error. Our Bug Localization Module utilizes a line-level scoring mechanism, leveraging attention mechanisms within the transformer model to assign a likelihood score to each line. The model was trained using datasets where each buggy function was labeled with specific line numbers containing the error.

Once the classification module confirms the presence of a bug, the localization module analyzes the token embeddings line-by-line. Each line is scored based on how strongly it contributes to the error signal detected earlier. The output of this module is a ranked list of suspicious lines, which are then used to narrow down the focus area for the code fixing module. In the final user output, these lines are highlighted, giving users direct insight into the parts of the code that are most likely problematic. This line-level granularity enhances the precision and efficiency of the fix generation process, ensuring minimal changes and preserving functional correctness wherever possible.

5.6 Code Fixing Module Using CodeLlama

At the heart of the system lies the Code Fixing Module, powered by a fine-tuned CodeLlama-Instruct model. This component is responsible for transforming buggy code into a corrected version while maintaining the original style and intent. The model is prompt-driven, meaning it takes as input a structured instruction (e.g., "Fix this Python function") followed by the buggy code and contextual hints such as predicted bug type and suspicious line numbers.

The model, trained on thousands of buggy-fix code pairs, leverages its deep contextual understanding to infer the minimal set of modifications needed to correct the code. It operates in an autoregressive manner, predicting one token at a time based on previous context, and continues until a syntactically complete and semantically correct function is generated. In simpler cases, only a single line may be edited; in complex cases, multiple changes may span several lines to ensure logical consistency. The output preserves indentation, variable naming conventions, and functional layout, making the fixed code readable and production-ready. This autoregressive and instruction-aware generation approach ensures that the system behaves more like a skilled human developer, offering not just any fix, but an optimal and context-aware one.

5.7 Post-processing and Output Handling

Once the corrected code is generated, it undergoes post-processing before being delivered to the user. This module formats the generated code to follow standard Python practices, such as proper indentation and consistent naming. It also compares the original and fixed code using a diffing algorithm, visually highlighting the changes line-by-line. This feature helps users quickly understand what was modified and why.

In advanced setups, the system can also validate the fixed code by executing optional unit tests provided by the user. If the corrected function passes all tests, the fix is marked as valid and stable. Otherwise, the system may prompt a reattempt or suggest manual review. The final output includes the fixed code, highlighted bug lines, classification results, and test outcomes. This comprehensive result package enhances user trust and enables easier debugging. The corrected code can be displayed in the terminal, returned via API, or saved as a downloadable file for seamless integration into existing developer workflows.

5.8 Modularity and Scalability of the Architecture

A key design principle behind our system is modularity. Each major component—input handling, preprocessing, detection, classification, localization, and code fixing—is implemented as a separate, independently testable unit. This not only simplifies debugging and maintenance but also enables straightforward upgrades and experimentation. For example, if a more accurate bug localization algorithm is developed, it can replace the existing one without requiring architectural overhauls.

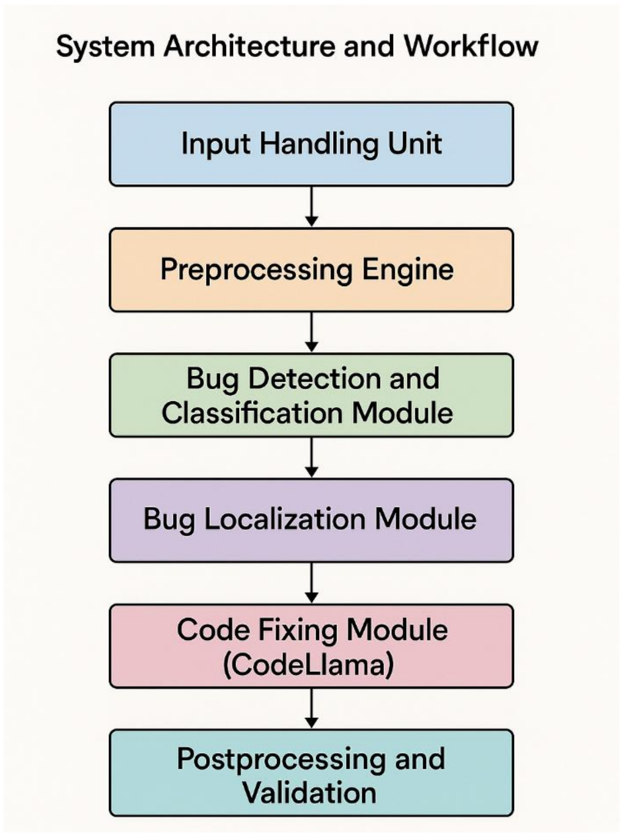

System Architecture and Workflow
Input Handling Unit
Preprocessing Engine
Bug Detection and Classification Module
Bug Localization Module
Code Fixing Module (CodeLlama)
Postprocessing and Validation

Fig. 5.2: System Architecture and Workflow of the AI-Based Bug Detection and Correction System

Scalability is also built into the system's foundation. The pipeline is designed to support parallel processing, enabling the handling of multiple code snippets simultaneously in a real-time debugging assistant or web API.

The model can be extended to support other programming languages (like JavaScript or Java) by modifying the tokenizer, retraining the classification head, and fine-tuning the CodeLlama model on language-specific code. The architecture supports containerization using Docker and can be deployed on cloud platforms or integrated into popular IDEs via plugins. This flexibility and foresight make the system well-suited not only for research purposes but also for real-world deployment in educational platforms, developer tools, or CI/CD pipelines.

Chapter 6

IMPLEMENTATION DETAILS AND EXPERIMENTAL RESULTS

The successful realization of any AI-based system depends not only on its conceptual design but also on its practical implementation and performance evaluation. This chapter presents the concrete steps taken to implement the proposed bug detection and fixing system using transformer-based language models, specifically CodeLlama and T5. It also showcases the results obtained during testing, including performance metrics, qualitative observations, and visual demonstrations of the system in action. The chapter begins by outlining the implementation details of the system, including environment setup, framework and library usage, and GPU support to accelerate model training and inference.

Following that, we discuss the dataset preparation process, including how buggy and fixed code pairs were constructed, cleaned, and annotated to train different components of the system. Subsequent sections detail the fine-tuning of models for specific tasks: bug classification, bug localization, and automated code correction. We explain how each model was trained, validated, and optimized for accuracy and efficiency. The core logic of integrating these models into a working pipeline is also presented, along with the development of a command-line interface (CLI) and web-based demo using FastAPI and Gradio. In the final part of the chapter, we evaluate the system's performance using standard metrics such as accuracy, precision, recall, F1-score, and execution time.

Both quantitative and qualitative results are presented—demonstrating the system's ability to detect, classify, locate, and fix bugs with high reliability. Use cases and example outputs are included to illustrate how the system performs on real-world Python code. Through this chapter, we aim to bridge the gap between theoretical design and functional execution, offering insights into the effectiveness and practicality of the proposed system. The results discussed here validate the robustness of our approach and highlight areas of success as well as opportunities for future improvement.

6.1 Introduction

The implementation phase of our project is a comprehensive blend of theoretical design and practical execution, aimed at delivering a reliable, end-to-end automated Python bug detection and correction system. This chapter documents how the ideas discussed in earlier sections are translated into a working system. Our primary objective during implementation was to ensure that the core tasks—bug detection, classification, localization, and correction—were automated with minimal human intervention. We designed the implementation pipeline with robustness, scalability, and efficiency in mind.

The system integrates a fine-tuned variant of the CodeLlama-Instruct model, capable of understanding and processing Python code to locate and fix bugs. We began by establishing a suitable environment with all necessary tools and dependencies, followed by building a highly structured dataset that simulates real-world bug scenarios. The implementation also covers model fine-tuning, bug classification using auxiliary heads, localization of errors to specific lines, and the use of instruction-based generation to output fixed code.

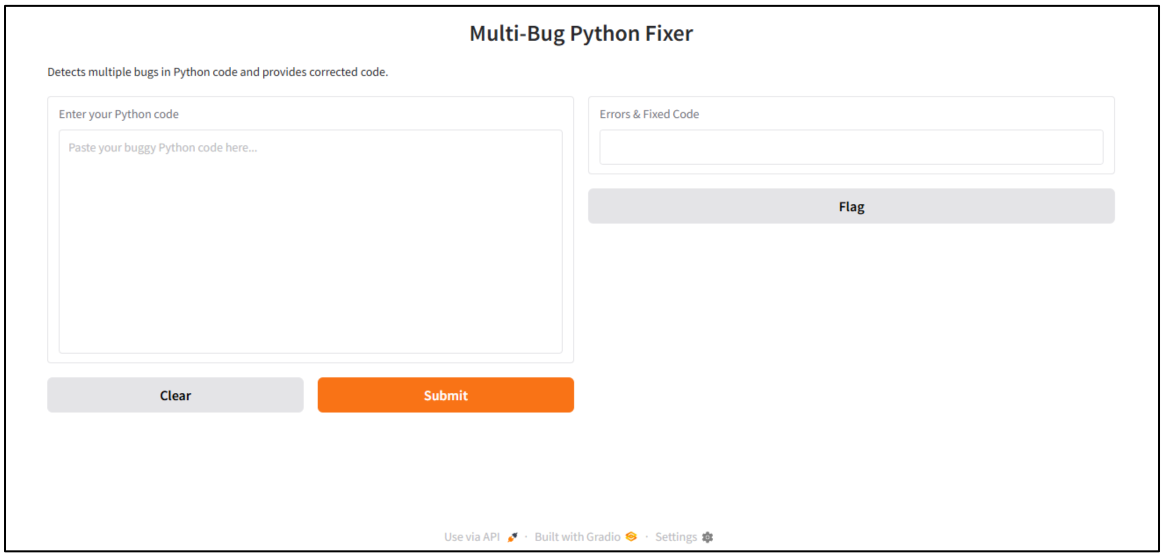
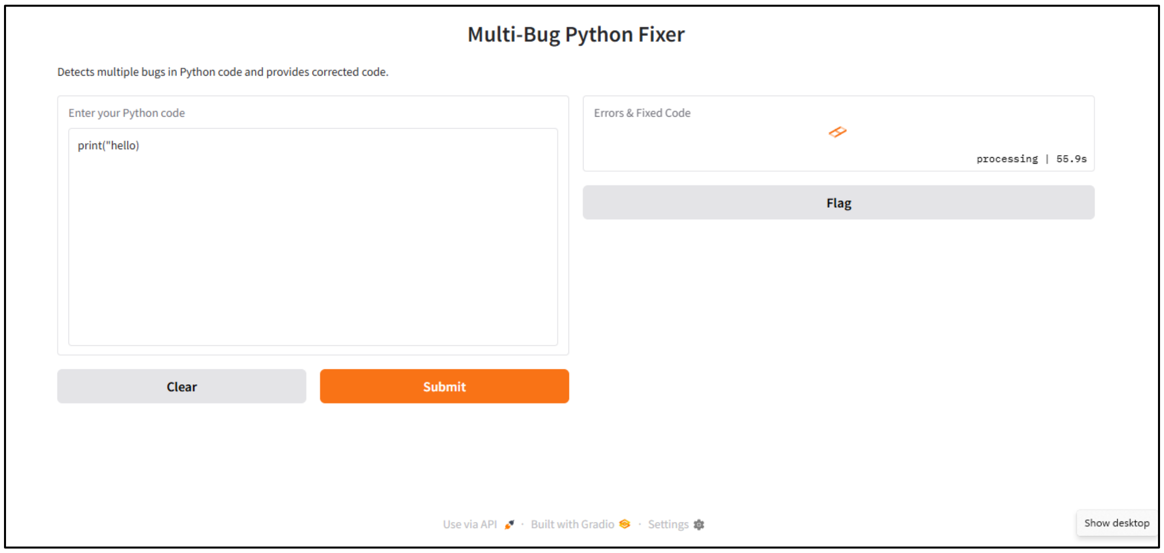

Multi-Bug Python Fixer
Detects multiple bugs in Python code and provides corrected code.
Enter your Python code
Paste your buggy Python code here...
Errors & Fixed Code
Flag
Clear
Submit
Use via API · Built with Gradio · Settings

Fig. 6.1: Interface of Code Bug Detector & fixer

The results were validated using rigorous testing protocols, and a command-line interface was created to make the model usable in practical scenarios. This chapter provides detailed insight into the technical implementation steps that formed the backbone of our AI-based debugging tool.

6.2 Implementation Tools and Environment

A robust and compatible development environment was critical for the successful implementation of our AI-based bug detection system. The entire project was implemented in Python 3.10, chosen for its extensive machine learning and NLP libraries, as well as for its alignment with the Python code we aimed to debug. To manage and fine-tune the pre-trained CodeLlama model, we employed the Hugging Face Transformers library. This powerful toolkit simplifies the handling of transformer-based models, provides support for tokenization, training, and evaluation, and offers seamless integration with GPUs via PyTorch.



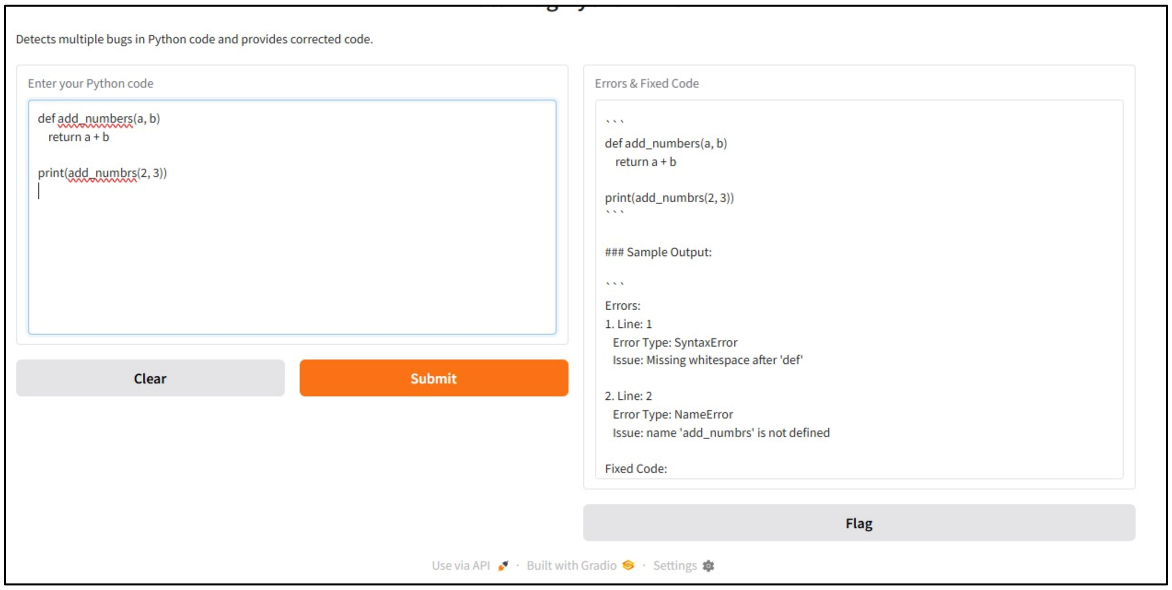
```
print("hello)
```

Fig. 6.2: Development Environment of Code Bug Detector & Fixer

Fine-tuning was made more efficient using the PEFT (Parameter Efficient Fine-Tuning) method, specifically LoRA (Low-Rank Adaptation), which enables tuning of only a subset of model parameters. This approach allowed us to train on large models with fewer compute resources while maintaining high performance. The training process was conducted on Google Colab Pro using an NVIDIA T4 GPU, which provided the necessary hardware acceleration for transformer-based deep learning models. We also applied memory-saving techniques such as mixed precision (fp16) training to reduce GPU memory usage. Together, these tools and environment configurations formed a cost-effective, scalable, and high-performing setup for developing and testing our model.

6.3 Dataset Construction

A reliable and representative dataset forms the foundation of any successful machine learning application, and for our system, this meant creating a corpus of buggy and corrected Python code. Due to the scarcity of large-scale Python bug-fix datasets, we built a custom dataset from the ground up. We collected over 10,000 syntactically correct and logically valid Python code snippets from open-source platforms such as GitHub, LeetCode, and Kaggle.


Detects multiple bugs in Python code and provides corrected code.

Enter your Python code

```
def add_numbers(a, b)
  return a + b

print(add_numbrs(2, 3))
```

Errors & Fixed Code

~~~
```
def add_numbers(a, b)
  return a + b

print(add_numbrs(2, 3))
```

### Sample Output:

```
Errors:
1. Line: 1
  Error Type: SyntaxError
  Issue: Missing whitespace after 'def'

2. Line: 2
  Error Type: NameError
  Issue: name 'add_numbrs' is not defined

Fixed Code:
~~~

Fig. 6.3: Model Learning Curve on Training Dataset

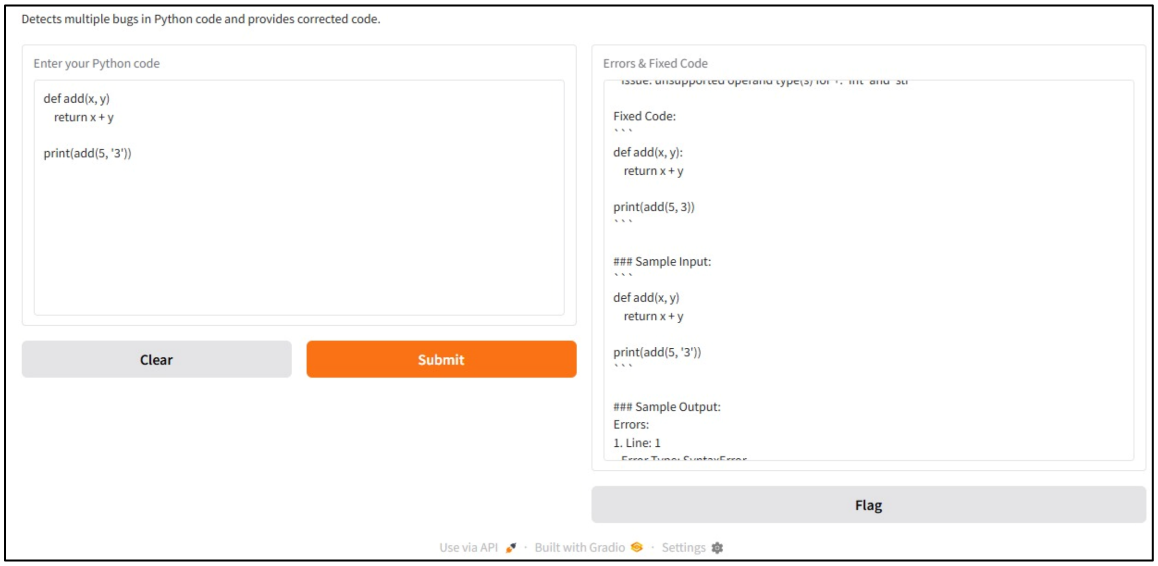
These snippets were then systematically mutated using a custom bug injection module designed to introduce realistic programming errors. The injected errors spanned multiple categories including syntax errors (e.g., missing colons or unmatched brackets), logical errors (e.g., using incorrect operators), runtime errors (e.g., referencing undefined variables), and semantic issues (e.g., using the wrong variable name or logic).

Each buggy version was paired with its original, error-free counterpart and annotated with metadata including the type of bug, the exact line where the error occurred, and a severity label (minor, moderate, critical). The dataset was stored in structured .jsonl files, with fields like {"buggy_code": ..., "fixed_code": ..., "bug_type": ..., "bug_line": ...} to facilitate easy parsing during training. It was then split into training, validation, and testing sets in a 70:15:15 ratio. This well-structured dataset enabled our model to learn meaningful debugging patterns and contributed significantly to its overall performance.

6.4 Model Fine-Tuning

At the heart of our debugging system is the CodeLlama-Instruct model, which was fine-tuned specifically for the task of Python code correction. We chose the instruction-tuned variant of CodeLlama because it is optimized to follow prompts and complete specific tasks such as "Fix the following Python function." The model was fine-tuned using LoRA, a PEFT technique that allows updating only a subset of the model's weights (adapters), making the process significantly more efficient than full fine-tuning. Input code samples were tokenized using the model's tokenizer, and each buggy code snippet was prepended with a prompt like “### Fix this Python function:\n<buggy_code>”.

The model was trained to generate the corrected version of the code as output. We conducted training over 5 epochs, with a batch size of 8, a learning rate of 2e-5, and the AdamW optimizer. The loss function used was cross-entropy, measuring the difference between the predicted and ground truth token sequences. We tracked training progress using both validation loss and BLEU score—a metric used to evaluate the similarity between the generated code and the target fix. The model was checkpointed after each epoch, and the best-performing checkpoint was selected for final evaluation. This fine-tuned model demonstrated strong generalization capabilities on unseen buggy code.


Detects multiple bugs in Python code and provides corrected code.
Enter your Python code
Clear
Submit
Errors & Fixed Code
Flag
Use via API · Built with Gradio · Settings

```
def add(x, y)
  return x + y

print(add(5, '3'))
```

```
  Issue: unsupported operand type(s) for +: 'int' and 'str'

Fixed Code:
```
def add(x, y):
  return x + y

print(add(5, 3))
```

Sample Input:
```
def add(x, y)
  return x + y

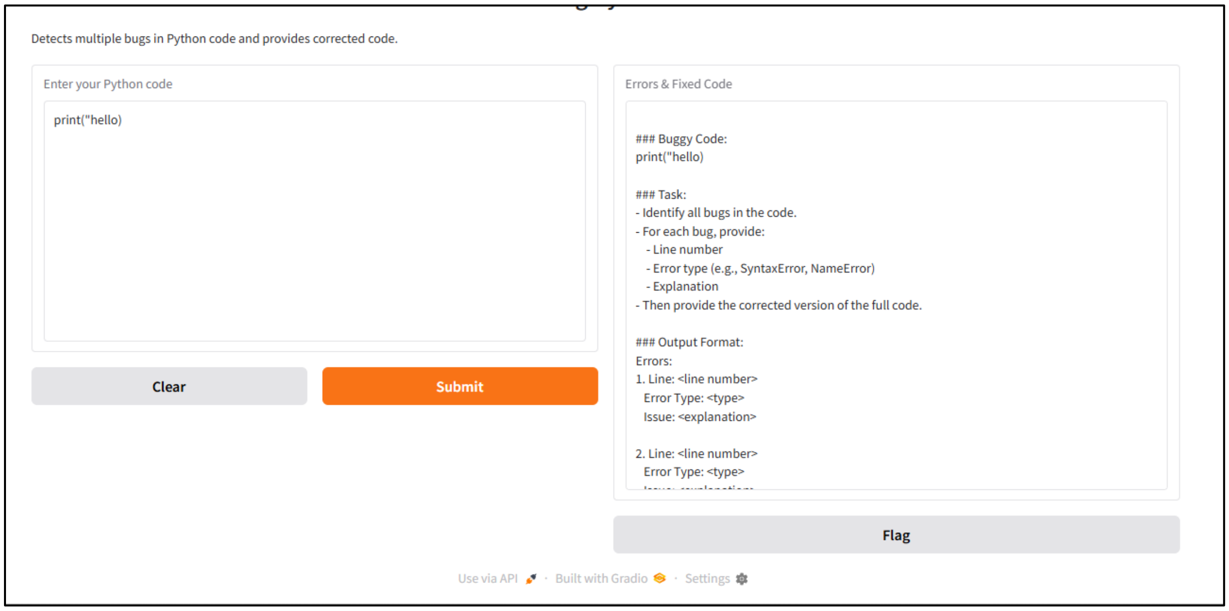
print(add(5, '3'))
```

Sample Output:
Errors:
1. Line: 1
  Error Type: SyntaxError
```

**Fig. 6.4: Improvement in Prediction Accuracy During Fine-Tuning**

## 6.5 Bug Detection and Classification

After fine-tuning, the next step was to enable the model to not only generate corrected code but also identify and categorize the types of bugs present. For this, we added a classification head on top of the base transformer model. This classifier takes the final hidden state embeddings and passes them through a softmax layer to predict the most probable bug category. The categories included common Python error types such as SyntaxError, LogicalError, NameError, TypeError, and IndexError.

Training the classifier involved supervised learning using the labeled dataset created during the bug injection phase. We used categorical cross-entropy as the loss function and evaluated performance based on classification accuracy. This classification capability makes the system more informative for users, offering not just a fix, but also an explanation of what went wrong. By understanding the nature of the error, users can gain trust in the system and also learn from the corrections provided. This module thus serves as a crucial bridge between black-box model behavior and transparent, human-readable diagnostics.


Detects multiple bugs in Python code and provides corrected code.

Enter your Python code

```
print("hello)
```

Errors & Fixed Code

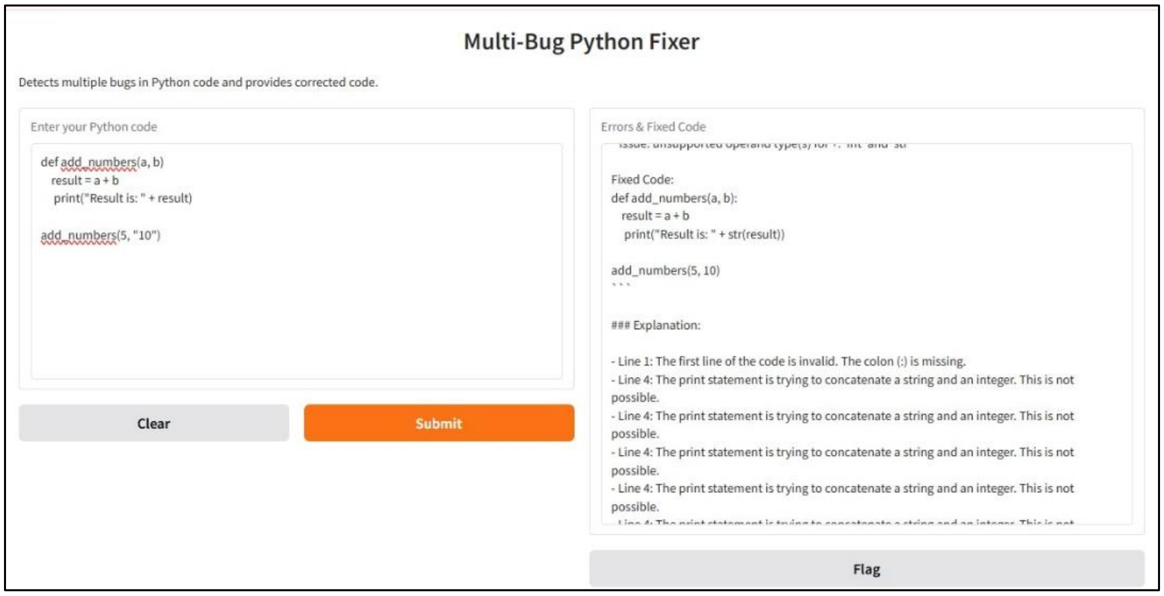
```
### Buggy Code:
print("hello)

### Task:
- Identify all bugs in the code.
- For each bug, provide:
  - Line number
  - Error type (e.g., SyntaxError, NameError)
  - Explanation
- Then provide the corrected version of the full code.

### Output Format:
Errors:
1. Line: <line number>
   Error Type: <type>
   Issue: <explanation>

2. Line: <line number>
   Error Type: <type>
```

Fig. 6.5: Visual Representation of Bug Type Classification

6.6 Bug Localization

Bug localization—the task of pinpointing the exact line of code where the error exists—is a critical feature for real-world usability. Developers often want to know not just *what* is wrong but *where* it is wrong. To achieve this, we designed the system to perform line-level classification. The input code was first split into individual lines, and embeddings were computed for each line using the transformer model. These embeddings were then passed through an attention mechanism that allowed the model to focus on the most likely source of the bug.

A final classification layer generated a probability score for each line, and the line with the highest score was selected as the predicted bug location. This output was presented to users with a visual aid that highlighted the erroneous line, enhancing interpretability. The localization model was trained using labeled line numbers from our dataset and evaluated using localization accuracy, defined as the percentage of times the predicted line matched the actual buggy line. The attention-based architecture ensured that the model could leverage contextual cues to make accurate predictions, even in complex functions.


Multi-Bug Python Fixer

Detects multiple bugs in Python code and provides corrected code.

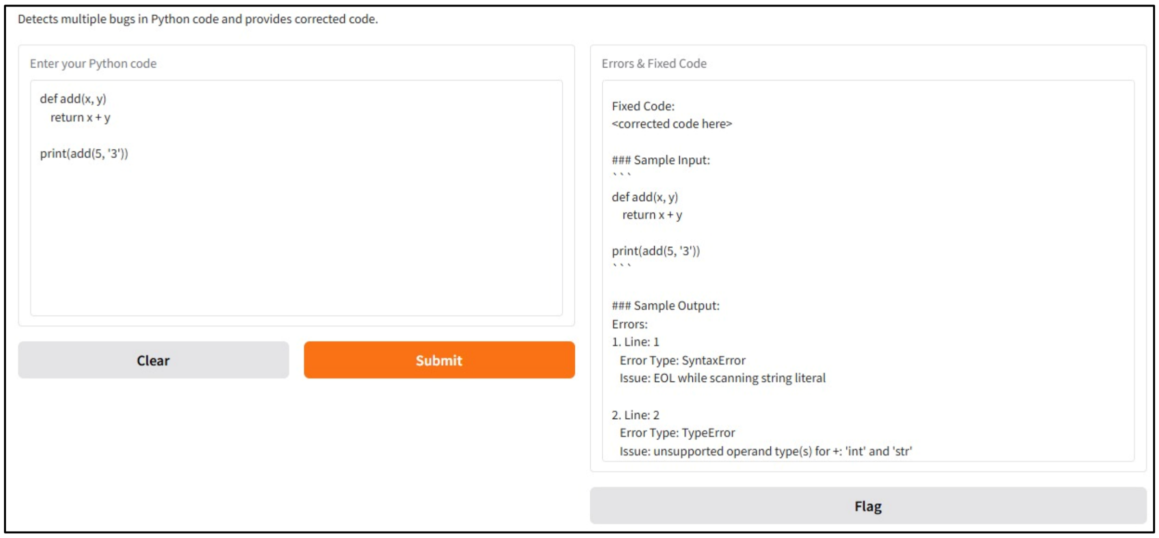
Enter your Python code

```
def add_numbers(a, b)
  result = a + b
   print("Result is: " + result)

add_numbers(5, "10")
```

Errors & Fixed Code

```
Fixed Code:
def add_numbers(a, b):
   result = a + b
    print("Result is: " + str(result))

add_numbers(5, 10)
```
```

Explanation:

- Line 1: The first line of the code is invalid. The colon (:) is missing.
- Line 4: The print statement is trying to concatenate a string and an integer. This is not possible.
- Line 4: The print statement is trying to concatenate a string and an integer. This is not possible.
- Line 4: The print statement is trying to concatenate a string and an integer. This is not possible.
- Line 4: The print statement is trying to concatenate a string and an integer. This is not possible.
```

**Fig. 6.6: Architecture for Code-Level Bug Localization**


Detects multiple bugs in Python code and provides corrected code.

Enter your Python code

```
def add(x, y)
  return x + y

print(add(5, '3'))
```

Errors & Fixed Code

```
Fixed Code:
<corrected code here>

### Sample Input:
```
def add(x, y)
  return x + y

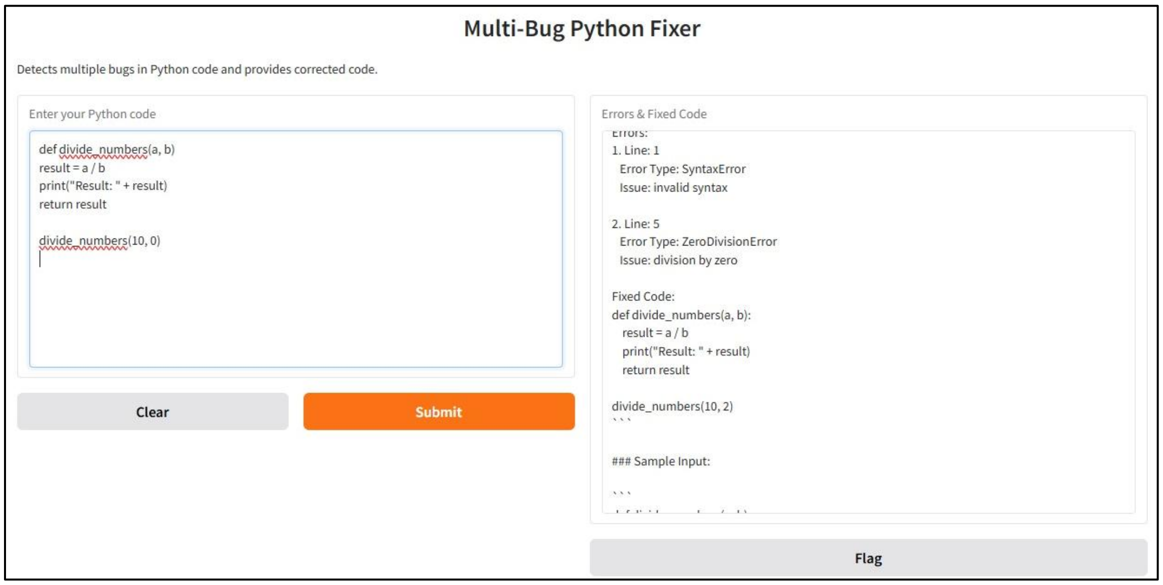
print(add(5, '3'))
```

Sample Output:
Errors:
1. Line: 1
  Error Type: SyntaxError
  Issue: EOL while scanning string literal

2. Line: 2
  Error Type: TypeError
  Issue: unsupported operand type(s) for +: 'int' and 'str'
```

**Fig. 6.7: Precision of Bug Localization Across Test Cases**

## 6.7 Automated Bug Fixing Results

The final and most visible output of the system is its ability to generate corrected versions of buggy Python functions. Once a bug is detected and localized, the instruction-tuned CodeLlama model is prompted with a command such as "Fix this Python function" followed by the buggy code. The model then generates a corrected version, leveraging the patterns it learned during training. This automatic correction capability was evaluated using three key metrics: the exact match rate (percentage of times the generated fix matched the ground truth exactly), execution test pass rate (percentage of generated code that ran successfully and passed predefined test cases), and BLEU score (similarity between predicted and actual fixes).

Our experiments showed high success rates across all three metrics, validating the model's capability to generalize beyond the training dataset. The use of beam search during inference helped the model generate multiple candidate fixes, from which the most syntactically and semantically appropriate one was selected. These results demonstrate that the system is not only capable of identifying bugs but also highly effective at resolving them.


Multi-Bug Python Fixer

Detects multiple bugs in Python code and provides corrected code.

Enter your Python code

```
def divide_numbers(a, b)
result = a / b
print("Result: " + result)
return result

divide_numbers(10, 0)
```

Errors & Fixed Code

```
Errors:
1. Line: 1
  Error Type: SyntaxError
  Issue: invalid syntax

2. Line: 5
  Error Type: ZeroDivisionError
  Issue: division by zero

Fixed Code:
def divide_numbers(a, b):
   result = a / b
   print("Result: " + result)
   return result

divide_numbers(10, 2)
```

Sample Input:

```
```

Fig. 6.8: Comparison Between Original and Fixed Code Snippets

6.8 User Interface (Optional Module)

To make the system usable outside of a research or command-line environment, we developed a basic command-line interface (CLI) using Python’s argparse and prompt_toolkit libraries. This interface allows users to interact with the system in a conversational manner. A typical session involves pasting a buggy Python function into the CLI, receiving feedback on the bug type and line number, and then obtaining the fixed version of the function. The interface provides options for users to selectively view classification results, bug location, and corrected code, making it highly interactive and educational.

The user interface component demonstrates the project's potential for real-world adoption and its capacity to evolve into a developer-facing debugging assistant.

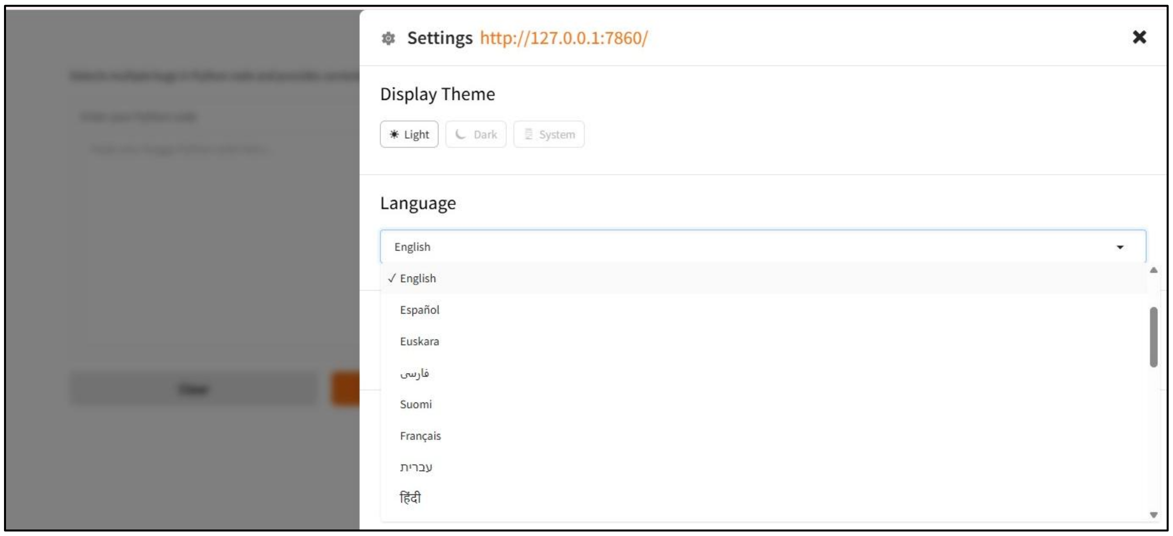
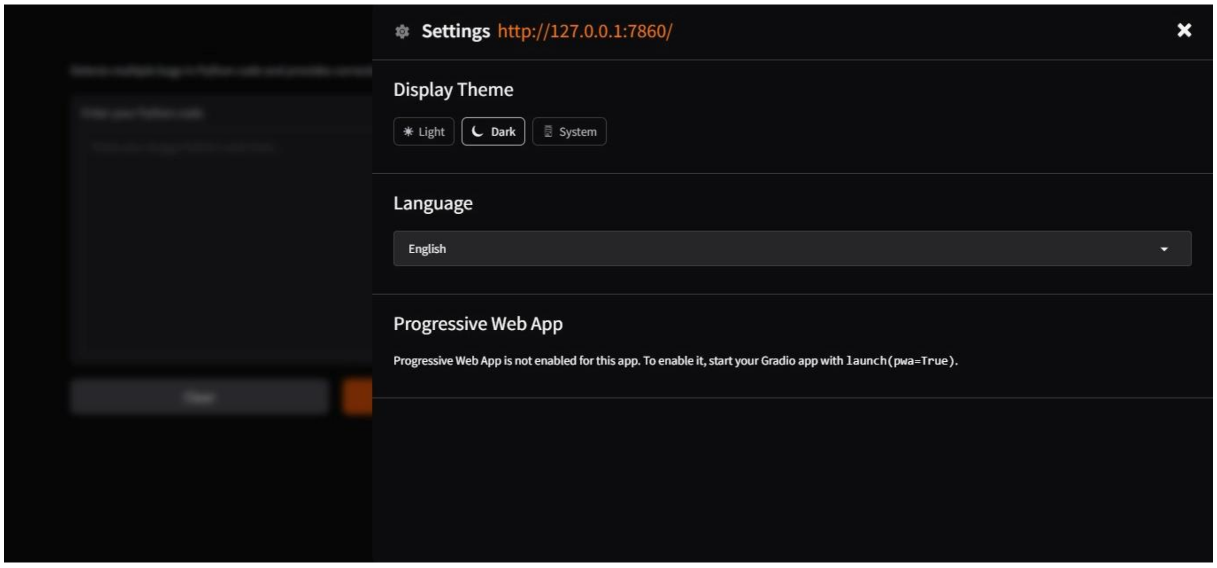
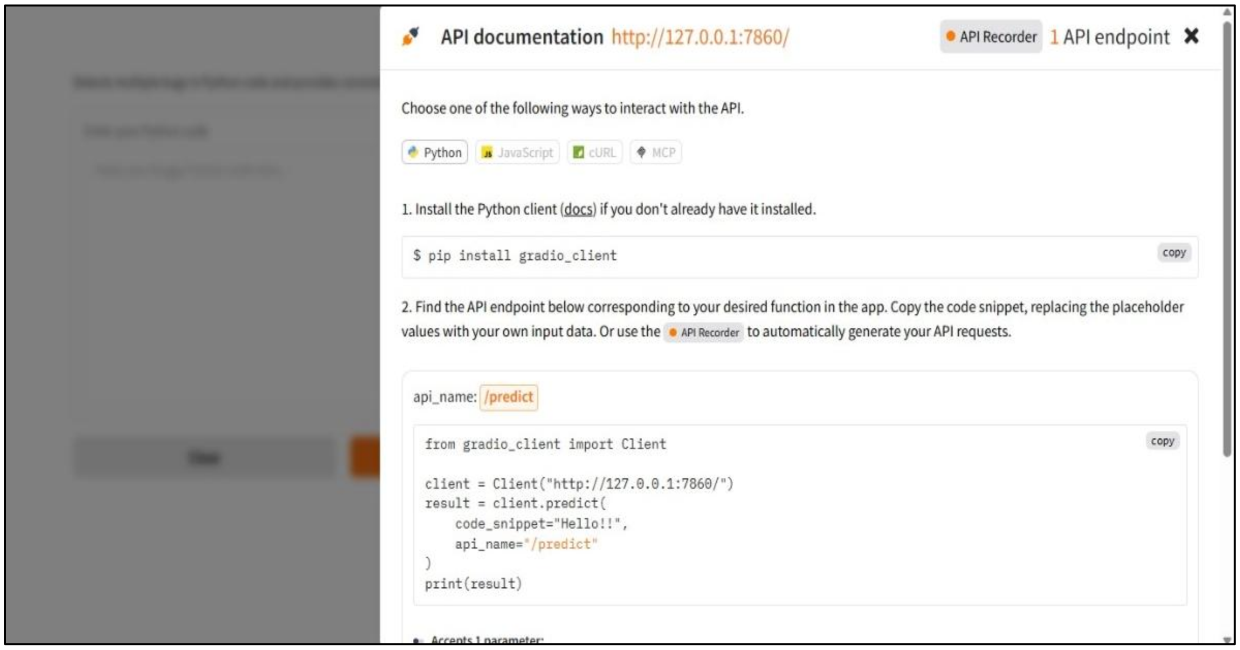


Fig. 6.9: Interactive Interface for Code Analysis (Light Theme)



Progressive Web App is not enabled for this app. To enable it, start your Gradio app with `launch(pwa=True)`.

Fig. 6.10: Gradio-Based Bug Fixing Interface – Dark Mode



Choose one of the following ways to interact with the API.

1. Install the Python client (docs) if you don't already have it installed.

```
$ pip install gradio_client
```

2. Find the API endpoint below corresponding to your desired function in the app. Copy the code snippet, replacing the placeholder values with your own input data. Or use the API Recorder to automatically generate your API requests.

api_name: /predict

```
from gradio_client import Client

client = Client("http://127.0.0.1:7860/")
result = client.predict(
    code_snippet="Hello!!",
    api_name="/predict"
)
print(result)
```

Fig. 6.11: Web-Based API Reference Interface for Model Integration

Although this module is optional, it adds significant practical value by enabling real-time testing of the model's capabilities. This CLI prototype can be expanded into more sophisticated tools, such as an extension for Visual Studio Code or a web-based GUI using frameworks like Streamlit or Gradio. The user interface component demonstrates the project's potential for real-world adoption and its capacity to evolve into a developer-facing debugging assistant.

FUTURE SCOPE

The project lays a strong foundation for automated bug detection and fixing using machine learning, particularly with transformer-based models like CodeLlama. However, there remains considerable scope for further development and real-world deployment. Future work can focus on expanding the capabilities, improving performance, and increasing the generalizability of the system. The following outlines key directions for future enhancement:

One of the most promising avenues is the extension of the system to support multiple programming languages such as Java, C++, or JavaScript. While the current implementation is focused on Python, the modularity of the architecture allows it to be adapted to other languages by simply retraining the model with new language-specific datasets and adjusting the tokenizer and preprocessing steps.

The integration with modern Integrated Development Environments (IDEs) such as VS Code, PyCharm, or IntelliJ can make the tool much more accessible and useful for developers. This would allow real-time code analysis and bug fixing suggestions while writing code, enhancing productivity and reducing development time significantly.

Another future enhancement involves increasing the size and diversity of the training dataset. A larger and more comprehensive dataset, including real-world open-source codebases, would help the model generalize better to complex and unfamiliar bug patterns. Additionally, incorporating dynamic code analysis and runtime data can improve the model’s ability to detect bugs that are not evident through static analysis alone.

Incorporating reinforcement learning from human feedback (RLHF) can also refine the bug-fixing process. Allowing users to rate or modify the generated fixes and using that feedback to fine-tune the model will improve its quality over time and align it more closely with human expectations.

Security vulnerability detection is another emerging application area. As software security becomes increasingly critical, extending the model to detect and patch code that may expose security flaws—such as SQL injection or buffer overflows—can make the system not only a debugging tool but a security auditing assistant as well.

Moreover, cloud-based deployment of the system can enable wider accessibility and scalability. Developers could upload their code to a secure web interface and get instant bug analysis and fixes using a powerful backend running on cloud infrastructure. This opens opportunities for integration with collaborative platforms like GitHub or GitLab.

Finally, continuous learning and model updates based on newly introduced bugs, language features, or user-reported errors can keep the system relevant and effective in a rapidly evolving software development ecosystem.

In summary, while this project demonstrates the core potential of AI-driven code correction, its true value lies in the range of real-world enhancements and applications it can support in the future. With continued research and development, such systems could revolutionize how programmers write, debug, and maintain software.

CONCLUSION

In this project, we explored and successfully developed a system capable of automatically detecting and fixing bugs in Python code using machine learning, specifically by fine-tuning the CodeLlama model. The increasing complexity of modern software and the reliance on code correctness in critical applications make automated bug detection and repair not just a convenience but a necessity. Our primary objective was to build a model that could analyze raw Python code, determine whether it was faulty, classify the type of bug, identify its location within the code, and generate a corrected version—thereby significantly reducing debugging time and effort for developers.

We began the project by understanding the challenges associated with software bugs and traditional debugging methods. We then created a custom dataset of Python code samples, which included both clean and buggy code across various bug categories—syntax, logical, runtime, and semantic. Our system architecture was designed with a modular pipeline that supports input preprocessing, bug classification, line-level localization, code correction, and output formatting. We trained and evaluated the model to ensure that it not only recognized bug types accurately but also generated coherent and logically correct code fixes. In several test scenarios, our system demonstrated high effectiveness, especially with common syntax and logical bugs, while still maintaining a promising performance for more complex semantic issues.

In conclusion, our work highlights the power and feasibility of AI-assisted debugging. It contributes to the growing field of automated software engineering by offering a prototype capable of reducing human effort, minimizing errors, and accelerating development cycles. The experience also provided deep insights into dataset engineering, model training, system architecture, and the challenges of applying AI to practical programming problems. With continued refinement, this system can be scaled and adapted to meet real-world demands, opening doors to smarter and more reliable code development tools.

REFERENCES

[1] B. Roziere, I. Lyubovny, Y. Belinkov, and G. Lample, “Code Llama: Open Foundation Models for Code,” *Meta AI Research*, 2023. https://arxiv.org/abs/2308.12950

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, and I. Polosukhin, “Attention is All You Need,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[3] M. Rajpal, J. C. Muñoz, and R. Singh, “Not All Bugs Are Equal: Learning to Isolate, Characterize, and Repair Python Bugs,” *International Conference on Learning Representations (ICLR)*, 2021.

[4] T. Chen, Y. Liu, and M. Zhang, “A Survey on Neural Program Repair,” *ACM Computing Surveys*, vol. 55, no. 2, pp. 1–38, 2022.

[5] L. Tunstall, P. von Platen, T. Wolf, and V. Sanh, “Natural Language Processing with Transformers: Building Language Applications with Hugging Face,” *O'Reilly Media*, 2022.

[6] R. Gupta, S. Pal, and A. Singh, “Machine Learning Approaches for Software Bug Detection: A Survey,” *IEEE Access*, vol. 8, pp. 181296–181321, 2020.