Bug Detection and Fixing

A Project Report

submitted in partial fulfillment of the requirements

of

Intel® Unnati Industrial Training 2025

by

Aishika Majumdar (bwubta22403@brainwareuniversity.ac.in)

Saikat Patra (bwubta22412@brainwareuniversity.ac.in)

Pritam Chakrabortty (bwubta22388@brainwareuniversity.ac.in)

Soumavo Acharjee (bwubta22401@brainwareuniversity.ac.in)

Under the guidance of

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

Acknowledgement

We take this opportunity to express our heartfelt gratitude to all those who supported and guided us throughout the development of our project, Bug Detection and Fix Recommendation System, undertaken as part of the Intel® Unnati Industrial Training 2025.

First and foremost, we would like to express our sincere thanks to **Dr. Shivnath Ghosh**, Head of the Department, CSE-AI, Brainware University, for his invaluable mentorship, encouragement, and insightful suggestions that greatly enriched our work. His guidance played a pivotal role in shaping this project and helping us stay focused on our goals.

We are also grateful to **Intel® Unnati Program** for providing us with the platform to explore the intersection of artificial intelligence and software engineering. The training, tools, and resources offered under this initiative were instrumental in the successful implementation of our system.

We extend our appreciation to the faculty members of **Brainware University** for fostering an environment that encourages innovation and research. Their constant support and constructive feedback inspired us to push the boundaries of our capabilities.

We are also thankful to our peers, friends, and families who supported us morally and motivated us throughout the project duration.

Lastly, we acknowledge the power of collaborative effort. As a team, we are proud of the mutual cooperation, commitment, and dedication that enable us to achieve our project objectives effectively.

Abstract

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

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

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

CONTENTS

Abstract 3

Chapter 1. Introduction 5-6

1.1 Problem Statement 5

1.2 Motivation 5

1.3 Objectives...5

1.4. Scope of the Project 6

Chapter 2. Literature Survey 7-9

2.1 Review of Relevant Literature ……………………………………..….7

2.2 Gaps and Limitations in existing solutions ……..…..…………..…..8

2.3 How our Project addresses these gaps…..…………………………...9

Chapter 3. Proposed Methodology 10-11

3.1 System design……………………………………………………..10

3.2 Requirement Specification ………….………………………….11

Chapter 4. Implementation and Results 12-14

4.1 Snapshot of result…………………………………………………12

Chapter 5. Discussion and Conclusion 15-16

5.1 Future scope………………………………………………………15

5.2 Conclusion……………………………………………………...16

References..17

1. Introduction

1.1 Problem Statement

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

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

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

1.2 Motivation

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

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

1.3 Objectives

The key objectives of this project are:

1. Data Collection & Preparation

- Gather code snippets one programming language.
- Label them as “buggy” or “bug-free” and include corrected versions for training.

2. Bug Detection Model

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

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

3. Fix Recommendation System
- Implement an AI-powered suggestion mechanism that proposes fixes for detected bugs.
- Ensure that recommendations are context-aware and improve code correctness.

4. Evaluation & Performance Metrics
- Use precision, recall, F1-score to assess the model's bug detection capability.
- Evaluate the accuracy and effectiveness of fix recommendations.

5. Usability
- Use an API for developers to input code and receive automated bug detection and fixes.
- Ensure the system generalizes across different coding styles and libraries.

1.4 Scope of the Project

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

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

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

2. Literature Survey

2.1 Review of relevant literature

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

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

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

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

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

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

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

2.2 Gaps and Limitations in Existing Solutions

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

Data Quality and Availability	Lack of High-Quality Labeled Datasets	• Most available datasets contain noisy, incomplete, or incorrect labels, leading to poor model generalization. • Bug-fix pairs from open-source repositories are often unstructured, making supervised learning challenging.
	Language and Framework Dependence	• Many existing solutions are language-specific, meaning a model trained on Python may not perform well on Java or C++. • Fix recommendation models often struggle with framework-specific bugs that require contextual understanding.
Model Performance Limitations	Poor Generalization to Unseen Code	• ML models often memorize patterns rather than truly understanding program logic. • Code structure varies widely between developers and projects, causing inconsistent predictions for real-world applications.
	Challenges in Understanding Logical Errors	• Syntax errors are easy to detect, but identifying semantic and logical bugs (e.g., infinite loops, incorrect algorithms) is still an open challenge. • Current AI models lack program execution simulation, leading to inaccurate bug detection.
	Fix Recommendations May Lack Context Awareness	• Suggested fixes are often syntactically correct but logically incorrect, as models focus more on surface-level patterns than deep program logic. • ML-based fixers struggle to differentiate between multiple possible correct fixes for the same bug.
Computational Challenges	High Computational Cost	• Transformer-based models like GPT require significant computational resources for training and inference. • Running these models in real-time environments, especially for large-scale projects, remains difficult.

	Inference Time and Latency Issues	• Real-time bug detection and auto-fixing require fast inference speeds, which is a challenge for large models. • The trade-off between model complexity and efficiency affects practical deployment.
Human Intervention Still Required	Reliance on Developers for Verification	• While AI can detect and suggest fixes, human review is still necessary to ensure correctness and security. • Many fixes require manual adjustments, making full automation difficult.
	Lack of Explainability	• Most ML models function as black boxes, meaning developers don't understand why a particular bug was flagged or why a fix was suggested. • This reduces trust and limits widespread adoption in production systems.
Security and Ethical Concerns	Vulnerability to Adversarial Attacks	• ML models can be tricked by adversarial code samples, leading to false positives/negatives in bug detection. • If attackers understand how the model works, they can intentionally bypass detection.
	Bias in Training Data	• Models trained on biased datasets may fail to detect certain types of bugs or suggest inappropriate fixes. • If not properly validated, ML-based fixers could introduce new security vulnerabilities instead of fixing them.

2.3 How our Project addresses these gaps

Our project aims to build upon these advancements by:

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

3. Proposed methodology

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

Step 1: Data Collection and Preprocessing

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

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

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

Step 3: Model Development

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

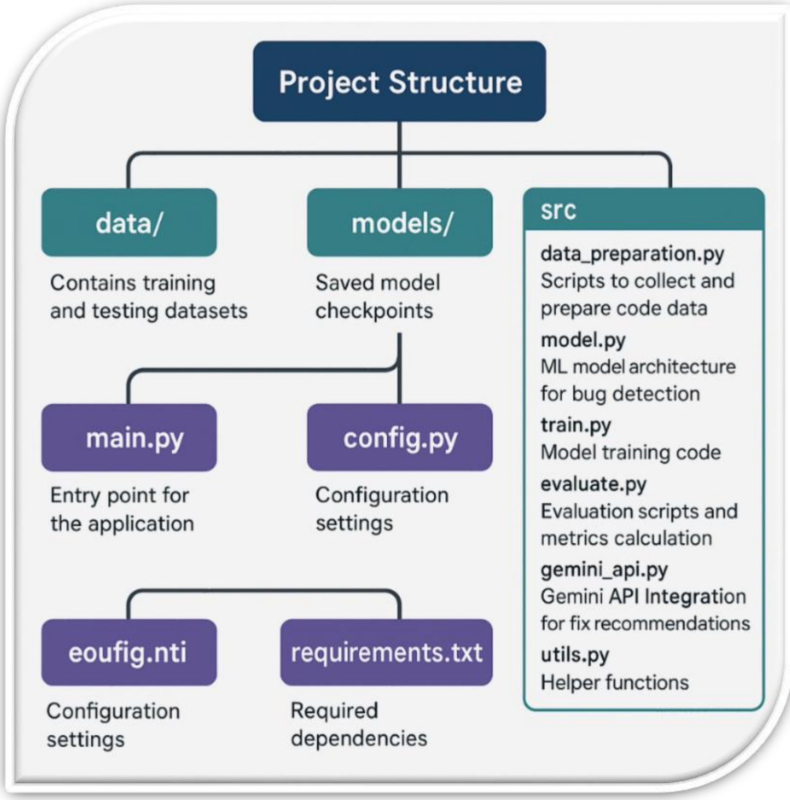
Step 4: Model Evaluation and Optimization - Evaluate model performance using:

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

Step 5: Utility

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

3.1 System Design



3.2 Requirement Specification

Software Requirements

1. Programming Language: Python
2. Frameworks & Libraries:
   - PyTorch / TensorFlow for deep learning
   - Scikit-learn for traditional ML
   - NLTK / Hugging Face Transformers for NLP models
3. Version Control: GitHub for collaborative development
4. Data Storage: Local Storage of at least 100 GB

Hardware Requirements

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

Security Measures:

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

4. Implementation and Results

4.1 Snapshot of Result

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

```
==================================================
ANALYSIS RESULTS FOR: sample_1.py
==================================================
BUG DETECTED (Confidence: 0.61)

--- RECOMMENDED FIX ---
def calculate_average(numbers):
    if not numbers:  # Handle empty list case
        return 0  # Or raise an exception: raise ValueError("Cannot calculate average of an empty list")
    total = 0
    for num in numbers:
        total += num
    return total / len(numbers)


data = [1, 2, 3, 4, 5]
avg = calculate_average(data)
print(f"The average is: {avg}")

data2 = []
avg2 = calculate_average(data2)
print(f"The average of an empty list is: {avg2}")

--- EXPLANATION ---
The original code had a potential bug: it would crash if an empty list was passed to the `calculate_average` function.  This is because `len(numbers)`
 would be 0, leading to a `ZeroDivisionError`.

The fix adds a check at the beginning of the function: `if not numbers: return 0`. This condition checks if the list is empty. If it is, the function
immediately returns 0, preventing the division by zero error.  Alternatively, you could raise a `ValueError` to explicitly signal that the input is in
valid.  I've added an example of this with `data2` and a second print statement to show both approaches.  Choosing between returning 0 and raising an
exception depends on the context and how you want to handle this situation in a larger program.  Returning 0 might be suitable if 0 is a meaningful de
fault value in your application, while raising an exception is better if an empty list represents an error condition.

--- METRICS ---
confidence: 0.6063430309295654
threshold: 0.5
==================================================
```

```
2025-04-04 22:31:59,150 - __main__ - INFO - Analyzing sample file: sample_3.py

==================================================
ANALYSIS RESULTS FOR: sample_3.py
==================================================
BUG DETECTED (Confidence: 0.60)

--- RECOMMENDED FIX ---
def merge_dicts(dict1, dict2):
    result = dict1.copy()
    result.update(dict2)
    return result

--- EXPLANATION ---
The original code was missing a colon at the end of the function definition line `def merge_dicts(dict1, dict2)`.  This is a syntax error in Python.
I've added the colon to correct the syntax.  Beyond that, the logic of copying `dict1`, updating with `dict2`, and returning the result is perfectly c
orrect for merging two dictionaries in Python where later values overwrite earlier ones (if keys overlap).

--- METRICS ---
confidence: 0.5970861315727234
threshold: 0.5
==================================================
```

```
2025-04-04 22:31:41,713 - __main__ - INFO - Analyzing sample file: sample_2.py

==================================================
ANALYSIS RESULTS FOR: sample_2.py
==================================================
BUG DETECTED (Confidence: 0.60)

--- RECOMMENDED FIX ---
def find_max(arr):
    if not arr:
        return None
    max_val = arr[0]
    for i in range(1, len(arr)):
        if arr[i] > max_val:
            max_val = arr[i]
    return max_val

--- EXPLANATION ---
The provided code appears correct.  It handles the edge case of an empty array by returning `None`. Otherwise, it iterates through the array, keeping
track of the maximum value encountered so far.  There are no apparent bugs in its logic or implementation.

--- METRICS ---
confidence: 0.6025850176811218
threshold: 0.5
==================================================
```

```
2025-04-04 22:32:11,206 - __main__ - INFO - Analyzing sample file: sample_4.py

==================================================
ANALYSIS RESULTS FOR: sample_4.py
==================================================
BUG DETECTED (Confidence: 0.60)

--- RECOMMENDED FIX ---
def search_list(items, target):
    for i in range(len(items)):
        if items[i] == target:
            return i
    return -1

--- EXPLANATION ---
The original code had a bug in its `range` function within the `for` loop.  `range(0, len(items)-1)`  incorrectly excluded the last element of the lis
t from the search.  If the target value happened to be the last element of `items`, the function would incorrectly return -1, indicating that the targ
et was not found.

The corrected code uses `range(len(items))`. This ensures that the loop iterates through all indices of the list, from 0 up to and including `len(item
s) - 1`, thus correctly searching the entire list.

--- METRICS ---
confidence: 0.6012831926345825
threshold: 0.5
==================================================
```

```
2025-04-04 22:32:15,040 - __main__ - INFO - Analyzing sample file: sample_5.py

==================================================
ANALYSIS RESULTS FOR: sample_5.py
==================================================
BUG DETECTED (Confidence: 0.60)

--- RECOMMENDED FIX ---
def calculate_discounted_price(price, discount_rate):
    discount = price * discount_rate
    final_price = price - discount
    return "The final price is " + str(final_price)

--- EXPLANATION ---
The original code had a type error.  The `final_price` variable is a number (float or integer, depending on the input `price`), but the `return` state
ment attempts to concatenate it with a string using the `+` operator. In Python, you cannot directly concatenate strings and numbers.

The fix involves converting the `final_price` to a string using the `str()` function before concatenation.  This allows the `+` operator to perform st
ring concatenation as intended.  The corrected code now returns a string that correctly represents the final price.

--- METRICS ---
confidence: 0.6024750471115112
threshold: 0.5
==================================================
```

```
2025-04-04 22:32:18,791 - __main__ - INFO - Analyzing sample file: sample_6.py

==================================================
ANALYSIS RESULTS FOR: sample_6.py
==================================================
BUG DETECTED (Confidence: 0.60)

--- RECOMMENDED FIX ---
import math

def calculate_area_and_circumference(radius):
    area = math.pi * radius**2
    circumference = 2 * math.pi * radius
    return area, circumference

def main():
    try:
        radius = int(input("Enter radius: "))
        if radius < 0:
            print("Radius cannot be negative.")
            return
        area, circumference = calculate_area_and_circumference(radius)
        print("The calculated area is:", area)
        print("The calculated circumference is:", circumference)
    except ValueError:
        print("Invalid input. Please enter an integer.")

if __name__ == '__main__':
    main()

--- EXPLANATION ---
The original code had several issues:

1. **Typographical Errors:**  There were several typos: `calculate_area_radius`, `Math`, `calclulate_area`, `mmain`, and `"Enter radius"` lacked a spa
ce. These were corrected to `calculate_area_and_circumference`, `math.pi`, `calculate_area_and_circumference`, `main`, and "Enter radius:".

2. **Incorrect Area Calculation:** The area calculation `Math.pi * rad ** 3` was wrong.  It used `rad` (which was undefined) and cubed the radius inst
ead of squaring it. The correct formula for the area of a circle is π * radius².  The code now correctly calculates the area.

3. **Missing `import math`:** The code was missing `import math` statement, necessary to use the `math.pi` constant. This has been added.

4. **Circumference Calculation Error:** The circumference calculation `2 * Pi 8 radius` was syntactically incorrect. It should be `2 * math.pi * radiu
s`. This has been corrected.
```

```
5. **Incorrect Return Value:** The `calculate_area_radius` function incorrectly tried to return `radius + area  area` which is syntactically incorrect
. It now returns both area and circumference as a tuple.

6. **Incorrect Type Handling in Output:** The original code attempted to concatenate a string and a number directly in the `print` statement,  which w
ould have resulted in a `TypeError`.  The fixed code uses commas to separate the string and the numerical value in `print` statements.

7. **Error Handling:** Added error handling using a `try-except` block to catch `ValueError` if the user enters non-integer input.

8. **Negative Radius Handling**: Added a check to ensure the radius is not negative, providing feedback to the user if it is.


The corrected code is now much more robust and accurately calculates both the area and circumference of a circle.

--- METRICS ---
confidence: 0.6042931079864502
threshold: 0.5
==================================================
```

```
2025-04-04 22:32:24,988 - __main__ - INFO - Analyzing sample file: sample_7.py

==================================================
ANALYSIS RESULTS FOR: sample_7.py
==================================================
BUG DETECTED (Confidence: 0.62)

--- RECOMMENDED FIX ---
def calculate_total(numbers):
    total = 0
    for i in range(len(numbers)):  # Corrected range to avoid index out of bounds
        total += numbers[i]
    return total  # Corrected typo

def reverse_string(s):
    reversed_str = ''
    for i in range(len(s)):
        reversed_str += s[len(s) - 1 - i]  # Corrected index calculation
    return reversed_str

def fetch_data_from_dict(data_dict, key):
    return data_dict[key]  # Corrected attribute access using square brackets

nums = [1, 2, 3, 4, 5]
total_sum = calculate_total(nums)
print(total_sum)  # Corrected typo

string = "debug"
reversed_result = reverse_string(string)
print(reversed_result)  # Corrected typo

data = {"name": "AI", "age": 3}
age = fetch_data_from_dict(data, "age")
print(age)  # Corrected typo

--- EXPLANATION ---
The original code had several issues:

1. **`calculate_total` function:** The `range(len(numbers) + 1)` in the `for` loop caused an `IndexError` because it tried to access an index beyond t
he last element of the list.  This was corrected to `range(len(numbers))`.  Additionally, there was a typo in the `return` statement ('totl' instead o
f 'total').
```

5. Discussion and Conclusion

5.1 Future Scope

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

1. **Support for More Programming Languages**

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

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

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

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

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

4. **Context-Aware Fix Recommendations**

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

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

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

6. **Explainable AI for Debugging**

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

7. **Expansion into Security Vulnerability Detection**

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

8. **Model Improvement with Reinforcement Learning**

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

5.2 Conclusion

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

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

6. References

1. Vassallo, L., et al. (2020). DeepCode: Learning to Detect Bugs and Vulnerabilities in Source Code.
2. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, Y., Gong, M., & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv preprint arXiv:2002.08155.
3. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A Survey of Machine Learning for Big Code and Naturalness. ACM Computing Surveys, 51(4), Article 81.
4. Tufano, M., Watson, C., Bavota, G., Di Penta, M., & Oliveto, R. (2019). An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. IEEE Transactions on Software Engineering, 45(2), 130–148.
5. OpenAI. (2021). Codex. Retrieved from https://openai.com/blog/openai-codex/ – This model has influenced the development of AI-powered code assistance tools.
6. Gupta, R., et al. (2017). Fault Localization and Repair: A Machine Learning Approach.
7. Svyatkovskiy, A., Sundaresan, N., Zhao, Y., & Fu, S. (2020). Intellicode Compose: Code generation using transformer.
8. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H., Kaplan, J. & Zaremba, W. (2021). Evaluating large language models trained on code.
9. Hugging Face. (n.d.). Transformers Documentation.
10. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems (NeurIPS).