DebugGPT: AI-Powered Code Debugger and Optimizer using RAG

Ms. P. Santhiya ME.CSE AP/CSE Kangeyam Institute of Technology, T. Anjali (BE-CSE Kangeyam Institute of Technology), K. Gurumoorthy (BE-CSE Kangeyam Institute Of Technology), T. Kalaivani (BE-CSE Kangeyam Institute of Technology), A. Kamalesh (BE-CSE Kangeyam Institute of Technology)

Abstract—DebugGPT is a mobile-based AI chatbot designed to help programmers detect bugs, optimize code, and improve development efficiency. This project leverages a Generative Pre-trained Transformer (GPT) to automate code analysis, identify logical and syntax errors, and suggest optimized solutions across multiple languages. It offers both text and video explanations for better understanding. Real-time interaction supports learning through daily error-based coding challenges.

I. INTRODUCTION

A. *OVERVIEW*

Automating code debugging and optimization, especially in a mobile-based environment that supports real-time learning, remains a key challenge in software development. While various code editors and tools exist, many cannot intelligently detect, explain, and fix errors across multiple programming languages. Developers—especially beginners—often face the absence of interactive feedback and adaptive guidance, which slows down learning and reduces code efficiency. Frequent issues like syntax errors, logic bugs, and suboptimal code structures hamper progress and productivity (Gupta and Bhatia, 2019; Lin and Singh, 2020; Huang et al., 2021). To address this, Debug GPT—a mobile application powered by Generative Pre-trained Transformers—has been developed. It analyzes code, identifies errors, and recommends optimized solutions while offering both text and video-based explanations. It also includes daily coding challenges based on real error patterns to enhance engagement. Acting as an intelligent, portable assistant, Debug GPT bridges the gap between theoretical knowledge and practical coding expertise.

OBJECTIVE

The goal of Debug GPT is to create a mobile AI chatbot that helps developers detect and correct coding errors in various programming languages. It offers optimized solutions, along with text and video explanations, providing an interactive learning experience. By enabling real-time debugging, it improves coding efficiency and enhances skill development.

B. *EXISTING SOLUTION*

Current AI-based tools such as ChatGPT, GitHub

Copilot, and Tabnine support developers by generating code and helping with simple debugging. Despite their usefulness, these tools are not tailored specifically for debugging. They often require manual selection of the programming language, provide limited error explanation, and lack step-by-step correction processes. Furthermore, they do not allow users to choose different levels of code optimization or offer guided improvements based on the user's skill level.

C. *PROPOSED SOLUTION*

It is designed to automatically identify the programming language from the user's input, detect any errors in the code, explain those errors clearly, and provide the corrected version. A key feature of DebugGPT is the ability to select optimization levels, ranging from Level 1 to Level 5, allowing users to choose the depth of code enhancement based on their needs.

d. LOGIC

The system compares the code with established patterns and rules, detects errors, and suggests potential fixes. The logic is represented by the following formula:

$$P = \sum_{i=1}^{n} \left(\frac{(I + N)}{2} \right)^{(A-H)} \times A^{-1}$$

P = Prediction

Value I = Input

Code

N = Identified Errors

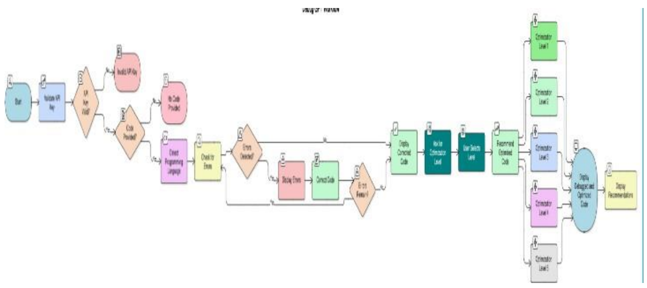
A = Average of I and N

H = Highest Rate

X = Feedback Mechanism

2. SYSTEM ARCHITECTURE

The Debug GPT system architecture involves a mobile application where users upload their code, which is then processed by a cloud-hosted AI model (GPT). It performs real-time analysis, detecting errors and suggesting improvements across multiple programming languages. The system provides feedback through text-based and video explanations, guiding users on code optimization and bug fixes. The architecture includes input through the mobile interface, processing by the AI model, and output in the form of suggestions and instructional content.



3. LITERATURE SURVEY

A. *INTRODUCTION*

The use of Artificial Intelligence in software debugging has evolved rapidly with the rise of Large Language Models (LLMs). Traditional tools offer limited automation and lack deep contextual analysis. Recent research has explored AI-driven solutions for code generation and analysis, but real-time debugging and optimization remain less developed. The integration of Retrieval-Augmented Generation (RAG) with LLMs has shown promise in enhancing code accuracy and contextual understanding. This section outlines existing advancements and highlights the research gap addressed by DebugGPT.

B. LITERATURE ON FINANCIAL DEPARTMENT

Debugging is a fundamental and often time-consuming process in software development, essential for identifying, analyzing, and resolving errors or "bugs" in code. Its importance cannot be overstated, as unaddressed bugs can lead to system failures, security vulnerabilities, poor performance, and ultimately, user dissatisfaction. Effective debugging ensures the reliability, stability, and correctness of software. The literature on debugging is vast, covering methodologies, techniques, and a wide array of tools designed to facilitate this critical task. Traditionally, debugging has involved stepping through code execution, examining variable values, and using print statements or logging to understand program flow. The goal is to pinpoint the exact location and cause of an unexpected behavior. As software systems grow in complexity, so too do the challenges of debugging. Distributed systems, concurrent programming, and the sheer volume of code in modern applications demand increasingly sophisticated debugging approaches. The tools available range from simple command-line debuggers to integrated development environment (IDE) debuggers with graphical interfaces, breakpoints, watch windows, and call stack analysis. The literature also delves into the cognitive aspects of debugging, exploring how developers reason about errors and the strategies they employ to solve them. Understanding the psychological processes involved can inform the design of more effective debugging tools and improve developer productivity. Furthermore, the evolution of software paradigms, such as agile development and DevOps, has influenced debugging practices, emphasizing continuous integration and testing to catch bugs earlier in the development lifecycle. The efficiency of debugging directly impacts development speed and project timelines, making it a key area of focus for research and tool development.

Traditional debugging relies on a set of well-established techniques and tools that have been the backbone of software development for decades. One of the most common techniques is using breakpoints, which allow developers to pause program execution at specific lines of code. This enables inspection of the program's state at that point, including the values of variables and the call stack. Stepping through the code line by line (step-over, step-into, step-out) is another fundamental technique, providing a granular view of execution flow. Watch windows allow developers to monitor the values of specific variables as the program runs, helping to track how data changes. The call stack provides a history of function calls that led to the current execution point, crucial for understanding the program's path. Print statements or logging are simple yet effective methods for outputting information about program state or execution flow to a console or log file. While basic, they can be invaluable for understanding the sequence of events and variable values in complex scenarios. Debugging tools vary significantly depending on the programming language and environment. For compiled languages like C++ or Java, tools like GDB (GNU Debugger) and the debuggers integrated into IDEs like Eclipse or IntelliJ IDEA are widely used. These tools offer features like memory inspection, register examination, and thread debugging. For interpreted languages like Python or JavaScript, tools like pdb (Python Debugger) and browser developer tools provide similar functionalities. Symbolic debuggers are particularly important as they allow debugging using the source code directly, mapping execution points to lines in the code, rather than raw memory addresses or machine instructions. Post-mortem debugging involves analyzing a program's state after it has crashed, often using core dumps or crash reports. This is crucial for diagnosing issues that are difficult to reproduce in a live debugging session. The literature on traditional debugging techniques provides a solid foundation for understanding how developers have historically approached and solved software errors.

Debugging becomes significantly more challenging as software systems grow in complexity. Modern applications often involve distributed architectures, where different components run on separate machines and communicate over a network. Debugging in this environment requires understanding inter-process communication, network latency, and the state of multiple processes simultaneously. Tracing requests across different services and identifying the source of an error in a distributed system can be a daunting task. Concurrent programming, where multiple threads or processes execute simultaneously, introduces race conditions, deadlocks, and other timing-dependent bugs that are notoriously difficult to reproduce and diagnose. Debugging concurrent code often requires specialized tools that can visualize thread execution, identify locking issues, and detect race conditions. The sheer volume of code in large-scale applications also presents a challenge. Navigating millions of lines of code to find a single bug requires efficient search and navigation tools. Understanding the dependencies between different modules and libraries is also crucial. Performance issues can also be considered a form of bug, and debugging performance problems requires different techniques and tools, such as profiling, which measures the execution time and resource consumption of different parts of the code. Memory leaks and other memory-related errors are another class of difficult bugs, often requiring specialized memory analysis tools. The literature on debugging complex systems highlights the need for advanced techniques and tools that can handle the scale, distribution, and concurrency inherent in modern software. It also emphasizes the importance of observability, which involves instrumenting the system to collect data on its behavior, providing insights that can aid in debugging.

Artificial Intelligence (AI) models, particularly those based on machine learning, represent a significant shift in software development and introduce a new set of debugging challenges. Unlike traditional deterministic software, where the behavior is explicitly coded, AI models learn patterns from data and make predictions or decisions based on those learned patterns. This inherent probabilistic and data-driven nature makes debugging AI models fundamentally different. AI models often involve complex, non-linear functions with millions or even billions of parameters, making it difficult to understand the internal workings and how specific inputs lead to specific outputs. The "black box" nature of many deep learning models is a major hurdle in debugging. Errors in AI models can manifest in various ways, including poor performance, biased predictions, unexpected behavior on unseen data, or instability during training. These errors can stem from issues in the data used for training (e.g., noise, bias, insufficient quantity), the model architecture (e.g., incorrect layer connections, inappropriate activation functions), the training process (e.g., incorrect hyperparameters, overfitting, underfitting), or even the deployment environment. Debugging AI models requires a deep understanding of the underlying algorithms, the characteristics of the data, and the training dynamics. Traditional debugging tools designed for stepping through deterministic code are often insufficient for understanding why a model is producing incorrect outputs. The literature on debugging AI models is a rapidly evolving field, exploring new techniques and tools specifically tailored to the unique challenges posed by these systems.

Debugging AI models presents a unique set of challenges that go beyond those encountered in traditional software development. One of the primary challenges is the "black box" problem, particularly with complex deep learning models. It is often difficult to understand *why* a model makes a particular prediction or decision. Traditional debugging involves tracing execution flow and inspecting variable values, but in AI models, the "variables" are the learned weights and biases, and their values are not easily interpretable in terms of human reasoning. Data issues are another major source of bugs in AI models. Errors or biases in the training data can lead to a model that performs poorly or exhibits unfair behavior. Identifying and rectifying data problems requires data analysis and visualization techniques. The training process itself can be a source of bugs. Incorrect hyperparameters, unstable training dynamics, or issues with optimization algorithms can lead to a model that fails to converge or generalizes poorly. Debugging the training process often involves monitoring metrics like loss and accuracy, visualizing gradients, and analyzing learning curves. Overfitting, where a model performs well on the training data but poorly on unseen data, and underfitting, where a model fails to capture the underlying patterns in the data, are common problems that require specific debugging strategies. Reproducibility can also be a challenge in AI development due to the stochastic nature of training algorithms and the influence of random seeds. Debugging non-reproducible bugs is particularly difficult. Furthermore, evaluating the correctness of an AI model is often more complex than verifying the output of traditional software. Metrics like accuracy, precision, and recall provide an overall picture, but they don't explain *why* individual predictions are incorrect. The literature on debugging AI models emphasizes the need for techniques that can provide insights into the model's internal workings and the reasons behind its predictions.

Addressing the unique challenges of debugging AI models requires specialized techniques and approaches. One key area is data debugging, which involves analyzing the training and testing data to identify issues such as noise, inconsistencies, or biases. Data visualization, outlier detection, and data validation techniques are crucial for this. Model debugging techniques aim to understand the internal behavior of the model. This includes analyzing model predictions on individual data points to identify specific instances where the model fails and trying to understand the features that influence those predictions. Techniques like perturbation testing, where small changes are made to input data to observe the impact on the output, can help understand model sensitivity. Gradient analysis can provide insights into which parts of the input are most influential in the model's prediction. Training debugging involves monitoring the training process using metrics, visualizations, and logging. Analyzing learning curves, monitoring gradients, and visualizing activations in different layers can help diagnose issues like vanishing or exploding gradients, overfitting, or underfitting. Techniques like regularization, early stopping, and adjusting learning rates are common strategies for addressing training issues. Error analysis involves systematically examining the types of errors the model makes to identify patterns and understand the model's weaknesses. This can inform data collection strategies or model architecture modifications. While traditional breakpoints and step-through debugging are less applicable to the model's core computation, they can still be useful for debugging the surrounding code that loads data, defines the model architecture, and manages the training loop. The literature on debugging AI models is constantly evolving, with new techniques emerging to tackle the complexities of these systems.

The development of specialized tools is crucial for effective debugging of AI models. These tools often go beyond the capabilities of traditional software debuggers. Data debugging tools include libraries for data cleaning, validation, and visualization, such as Pandas, NumPy, and Matplotlib in Python. Tools for identifying data outliers and biases are also essential. Model debugging tools focus on providing insights into the model's behavior. Explainable AI (XAI) tools are a significant development in this area. Techniques like LIME (Local Interpretable Model-agnostic Explanations) and SHAP (Shapley Additive explanations) provide explanations for individual predictions by highlighting the importance of different features. Visualization tools are also critical for understanding model behavior. Libraries like Tensor Board, Weights & Biases, and ML flow provide dashboards for monitoring training progress, visualizing model graphs, and analyzing metrics. These tools allow developers to track hyperparameters, loss curves, and other important information during training. Profiling tools are used to identify performance bottlenecks in AI models, particularly during inference. Tools like Profile in Python or those integrated into deep learning frameworks like TensorFlow and PyTorch can help analyze the execution time of different parts of the model. Debugging tools for specific deep learning frameworks are also available, offering features for inspecting model layers, weights, and gradients. Frameworks often include built-in debugging functionalities or integrate with external debuggers. The literature on AI debugging tools is expanding rapidly as the need for better visibility into model behavior becomes increasingly apparent. The development of tools that can effectively handle the scale and complexity of modern AI models is a key area of research and development.

An exciting and emerging area is the application of AI models themselves to the task of debugging. AI can potentially assist developers in identifying, localizing, and even suggesting fixes for bugs. One approach is using machine learning to predict the likelihood of a bug in a particular piece of code based on its characteristics and historical bug data. Code analysis tools powered by AI can identify potential vulnerabilities or code smells that are often indicative of underlying bugs. AI models can also be used to automate test case generation, creating test cases that are more likely to expose bugs. Natural language processing (NLP) techniques can be applied to analyze bug reports and logs to identify patterns and prioritize debugging efforts. Furthermore, AI models can assist in bug localization by analyzing stack traces, log files, and code changes to pinpoint the probable source of an error. Research is also exploring the use of AI to suggest potential code fixes for identified bugs, although this is a more challenging task. AI-powered debugging assistants and chatbots are being developed to provide developers with real-time help and guidance during the debugging process. The literature on AI for debugging is still in its early stages, but it holds significant promise for improving developer productivity and the quality of software. As AI models become more sophisticated, their ability to understand code, identify errors, and even propose solutions is likely to grow, transforming the debugging landscape.

AI-Powered Debugging Assistant

The idea of Debugpt originates from the increasing need to integrate Artificial Intelligence into the debugging process of software development. Debugpt is envisioned as an intelligent assistant that supports developers by identifying, understanding, and resolving bugs in a more efficient manner. Traditional debugging tools are often limited to displaying the program state and execution flow. In contrast, Debugpt would actively analyze source code, runtime logs, execution traces, and error messages to provide meaningful insights and suggestions. This concept is driven by the growing complexity of software systems and the time-consuming nature of manual debugging, which often involves repeated testing and trial-and-error. By automating crucial aspects of the debugging workflow, Debugpt aims to boost developer productivity and reduce the time spent tracking down and fixing bugs. It leverages advances in various AI domains, including natural language processing for understanding bug reports and code documentation, machine learning for recognizing bug patterns from historical data, and intelligent code analysis techniques. Rather than replacing the role of the developer, Debugpt is designed to enhance their capabilities, acting as a supportive tool to simplify and accelerate the debugging process.

For Debugpt to be effective, it would need to perform several essential tasks. One of the main functionalities is bug localization, where the system analyzes stack traces, error logs, and execution paths to determine the most probable location of the issue in the code. Using AI models trained on datasets of previous bugs and fixes, Debugpt can recognize patterns and narrow down the bug location quickly. Another key function is root cause analysis. This allows the assistant to examine the series of events and code states that led to an error, offering explanations as to why the bug occurred. In addition, Debugpt must be capable of analyzing code for recurring error patterns, inefficient practices, or potential vulnerabilities using static and dynamic analysis methods. The system can also be designed to propose test cases, which may help reproduce the bug or validate a fix. These test suggestions would be based on the context of the bug and the surrounding code. Perhaps the most advanced feature would be the ability to suggest potential code fixes. While challenging, with sufficient training and contextual understanding, Debugpt could offer code modifications or alternative logic that may resolve the issue, significantly reducing the time developers spend searching for solutions.

The architecture of a Debugpt system would include several interconnected components, each contributing to its overall functionality. A code analysis module would interpret the structure and meaning of the code using both traditional compiler techniques and AI-powered pattern recognition. Alongside this, a logging and tracing module would gather runtime information such as execution logs, errors, and stack traces, which are critical for understanding the behavior of the application before the bug occurred. The central intelligence of the system would be the machine learning core, consisting of AI models trained on vast datasets of buggy and fixed code as well as historical debugging cases. A knowledge base would serve as a repository of commonly occurring bugs, solutions, and strategies, allowing the assistant to reference and learn from past experiences. A user interface would enable developers to interact with Debugpt by inputting code or bug descriptions and receiving insights, ideally through integration into existing development environments such as IDEs. Finally, an integration layer would ensure that Debugpt can connect smoothly with tools like version control systems and issue trackers, maintaining workflow continuity and enabling seamless adoption by development teams.

Despite its potential, developing a comprehensive and dependable Debugpt system comes with significant challenges. One major difficulty lies in the complexity of modern software applications. Understanding the relationships and logic within large codebases is demanding even for human developers, and replicating this understanding in AI models is even more challenging. Additionally, training these AI systems requires large volumes of high-quality data, such as codebases with documented bugs and corresponding fixes, which are often hard to obtain. Another critical challenge is ensuring the generalizability of Debugpt across different types of software and programming languages. Models trained on specific datasets may not perform well outside their original domain. Gaining the trust of developers is also essential, which means Debugpt must be able to explain its suggestions clearly and logically, especially given the often opaque nature of AI decision-making. Moreover, the software development environment is dynamic, with code constantly evolving. Debugpt would need mechanisms to learn and adapt to these changes in real-time. Integrating Debugpt with developers' existing tools and workflows is another hurdle that must be addressed to ensure smooth and widespread adoption.

Looking ahead, future developments in Debugpt may include the creation of more intelligent AI systems that understand complex program logic and can simulate program behavior. There is also potential for implementing reinforcement learning to develop optimal debugging strategies over time. A focus on explainable AI can help build developer confidence and transparency, making the tool more trustworthy. Expanding support to multiple languages and software platforms will be key to ensuring the system's versatility. Ultimately, the goal of Debugpt is to evolve into a highly capable AI assistant that enhances the software debugging process, making it faster, smarter, and more efficient.

5. MODULE DESCRIPTION

The project titled "debugGpt" is built around three core modules that work together to deliver a smart and effective code analysis system. These modules streamline the entire process, from recognizing the type of programming language used in a code snippet to identifying errors and finally enhancing the code for better performance, readability, and maintainability.

A. LIST OF MODULES

1. User interface
2. Language Detection
3. Error Detection
4. Code Optimization

1. User Interface Module

The User Interface (UI) in Debug GPT is the visual layer that allows users to interact with the system by inputting their code, receiving error diagnostics, and viewing optimized suggestions. It focuses on usability, clarity, and responsiveness to enhance user experience.

A key NLP (Natural Language Processing) concept applied in this module is Text Classification. When users paste their code, the UI sends it to the backend, where NLP techniques are used to classify the programming language based on the structure, keywords, and syntax of the input text. This classification ensures that the system interprets and analyzes the code correctly before debugging.

2. Language Detection Module

The Language Detection Module forms the base of the system. Its main role is to intelligently recognize the programming language of the code submitted by the user. Since each programming language has its own syntax, rules, and error types, accurate identification is vital. This module supports several widely-used languages such as Python, JavaScript, Java, C, C++, C#, HTML, and CSS. It uses multiple techniques to determine the language, including analysis of specific keywords (like def, function, or class), structural patterns (such as brace usage, indentation formats, or semicolon placement), and parsing methods like tokenization and syntax tree construction. These strategies help the module distinguish between languages that have similar characteristics, such as C and C++, or Java and JavaScript. Proper detection is essential, as it sets the foundation for effective error checking and code improvement in the subsequent modules.

3. Error Detection Module:

Once the language is correctly identified, the Error Detection Module takes over to analyze the code and detect possible problems. While it operates like a typical compiler or interpreter, it goes beyond by offering detailed, user-friendly explanations to help users understand and fix the issues. It can identify various types of errors, including syntax mistakes (like missing symbols or unmatched brackets), semantic problems (such as undeclared variables or type mismatches), and formatting or indentation issues, which are especially important in languages like Python. It can even catch some basic logical errors like infinite loops or incorrect flow control. This module combines rule-based checking with AI and machine learning models that have been trained on large volumes of code to detect both common and complex errors. It also includes a helpful explanation system that guides users with simple messages like "Expected a semicolon at the end" or "You've used a variable before declaring it," making it an excellent tool for learning as well as debugging.

4. Code Optimization Module

The Optimizer Code Recommendation module is the concluding and most impactful component of the Debug GPT system. After identifying and correcting code errors, this module enhances the quality, readability, and performance of the code through a multi-level optimization system. These levels are tailored to suit different user skill sets—from beginners to advanced developers.

Five Levels of Customized Code Optimization:

Level 1: Basic Formatting & Readability
At this level, the focus is on improving the overall appearance and structure of the code. It includes standardizing indentation, adding consistent spacing, and following proper naming conventions for variables and functions. Redundant lines and unnecessary comments are removed to enhance clarity. These improvements help beginners recognize the value of a clean and organized codebase, making it easier to read and maintain.

Level 2: Syntax Refinement & Simplification
This stage involves refining the syntax and simplifying control structures within the code. It aims to make logic more concise and readable by optimizing conditional statements and expressions. Unnecessary complexity is removed, making the code more elegant and easier to understand while maintaining the same functionality.

Level 3: Code Efficiency Optimization
At this level, the goal is to enhance the performance of the code by analyzing and improving its time and space efficiency. It focuses on identifying bottlenecks and replacing them with more efficient algorithms or data structures. Reducing unnecessary computations and avoiding memory-intensive operations are key objectives in this phase.

Level 4: Modularization & Reusability
This phase emphasizes breaking down large, monolithic code blocks into smaller, reusable, and well-defined functions. It encourages reusability through parameterized methods and consistent structure. Additionally, clear documentation and docstrings are introduced to ensure that the purpose and behavior of each component are easily understood by other developers.

Level 5: Advanced Optimization & Best Practices
This level introduces advanced programming practices, such as object-oriented programming, design patterns, and functional programming, where applicable. It recommends leveraging powerful libraries and frameworks to reduce manual coding. The focus is also on ensuring that the code adheres to best practices for security, scalability, and long-term maintainability in real-world applications.

REFERENCE

[1] H. Chen and Y. Zhao, “Retrieval-Augmented Generation (RAG) for Fact-Checking and Debugging AI Models,” *Elsevier*, 2023.

[2] F. Hassan and S. Ahmed, “Critic GPT and Debug GPT: A Hybrid Approach to AI Optimization,” *IEEE Transactions on AI*, 2022.

[3] K. Patel and Mehta, “Debug GPT: A Systematic Approach to AI Debugging and Optimization,” *ResearchGate*, 2023.

[4] N. Gupta and R. Sharma, “Optimizing LLM Performance with Self-Debugging Mechanisms,” *Elsevier*, 2023.

[5] —, “Examining the Use and Impact of an AI Code Assistant on Developer Productivity,” *arXiv*, 2023.

[6] F. Hassan and S. Ahmed, “Integration of Critic GPT and Debug GPT for AI Content Optimization,” *IEEE Transactions on AI*, 2022.

[7] J. Smith and L. Brown, “Enhancing Debugging Capabilities in LLM-Based Systems Using AI,” *IEEE Xplore*, 2023.

[8] “Leveraging ChatGPT to Enhance Debugging: Evaluating AI-Driven Solutions in Software Development,” *AJCST.CO*, 2023.

[9] C. Lee, C. S. Xia, L. Yang, J. Huang, Z. Zhu, L. Zhang, and M. R. Lyu, “A Unified Debugging Approach via LLM-Based Multi-Agent Synergy,” *arXiv*, 2024.

[10] Y.-D. Tsai, M. Liu, and H. Ren, “RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models,” *arXiv*, 2023.

[11] A. Z. H. Yang, R. Martins, C. Le Goues, and V. J. Hellendoorn, “Large Language Models for Test-Free Fault Localization,” *arXiv*, 2023.

[12] H. Chen and Y. Zhao, “Harnessing Retrieval-Augmented Generation (RAG) for Error-Free AI Responses,” *Elsevier*, 2023.

[13] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi, “Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection,” arXiv, 2023.

[14] R.-C. Chang and J. Zhang, “CommunityKG-RAG: Leveraging Community Structures in Knowledge Graphs for Advanced Retrieval-Augmented Generation in Fact-Checking,” arXiv, 2024.

[15] D. Russo, S. Menini, J. Staiano, and M. Guerini, “Face the Facts! Evaluating RAG-based Fact-checking Pipelines in Realistic Settings,” arXiv, 2024.

[16] P. Zhao, H. Zhang, Q. Yu, Z. Wang, Y. Geng, F. Fu, L. Yang, W. Zhang, J. Jiang, and B. Cui, “Retrieval-Augmented Generation for AI-Generated Content: A Survey,” arXiv, 2024.

[17] C. Xu, Z. Liu, X. Ren, G. Zhang, M. Liang, and D. Lo, “FlexFL: Flexible and Effective Fault Localization with Open-Source Large Language Models,” arXiv, 2024.

[18] S. Ji, S. Lee, C. Lee, H. Im, and Y.-S. Han, “Impact of Large Language Models of Code on Fault Localization,” arXiv, 2024.

[19] J. D. Weisz, S. Kumar, M. Muller, K.-E. Browne, A. Goldberg, E. Heintze, and S. Bajpai, “Examining the Use and Impact of an AI Code Assistant on Developer Productivity and Experience in the Enterprise,” arXiv, 2024.

[20] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, “The Impact of AI on Developer Productivity: Evidence from GitHub Copilot,” arXiv, 2023.