International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 04 | April - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

AI CODE DEBUGGING ASSISTANT

Mrs. R. Kalayarasi, BAKTHAPRIYAN M, DARSHAN S, HIBA FATHIMA N

Assistant Professor, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, Coimbatore, Tamil Nadu, India.
UG Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, Coimbatore, Tamil Nadu, India.

ABSTRACT

The AI Code Debugging Assistant revolutionizes the software debugging workflow, offering an autonomous and reliable solution for developers. Designed with a focus on safety and efficiency, the agent leverages cutting-edge technologies such as generative AI (Google Gemini) for code patching, Abstract Syntax Tree (AST) analysis for high-context understanding, and secure Docker sandboxing for automated patch verification. By addressing the critical bottleneck of the manual "run-crash-fix-repeat" cycle, the agent fosters a more productive and streamlined development process. This paper explores the architecture, design principles, and key components of this autonomous system, highlighting its significant impact on developer productivity and its broader implications for self-healing software and automated CI/CD pipelines.

Keywords

Autonomous Systems, Generative AI, Automated Debugging, Secure Sandboxing, Developer Productivity.

I. INTRODUCTION

In today's fast-paced digital era, software development velocity is no longer a luxury but a fundamental requirement for technological and business success. For software developers, traditional debugging methods—a manual, reactive, and time-consuming "run-crash-fix- repeat" cycle—often fall short of meeting modern demands. The AI Code Debugging Assistant was developed to address this critical bottleneck, offering an autonomous platform that empowers developers to resolve runtime errors effortlessly, efficiently, and reliably.

The agent, which embodies the principles of autonomous "self-healing" software, integrates a suite of advanced technologies. These include real-time crash detection (via subprocess monitoring), high-context error analysis (using Abstract Syntax Trees), generative AI code patching (using Google Gemini), and, most critically, a "Trust, but Verify" secure sandboxing system (using the Docker SDK). Unlike traditional static analyzers, which cannot find runtime bugs, or passive AI assistants, which place the full burden of testing on the developer, this agent provides a complete, end-to-end, verified solution.

This paper discusses the conception, modular architecture, and implementation of the AI Code Repair Agent. We explore its role in breaking down the barriers of manual debugging and its potential to set new standards for automated software engineering. Furthermore, we examine the agent's impact on enhancing developer productivity, its application in automated CI/CD pipelines, and its ultimate contribution to a future of more resilient, reliable, and autonomous software systems.

II. LITERATURE REVIEW:

In today's fast-paced digital era, software development velocity is no longer a luxury but a fundamental requirement for innovation and business success. For software developers, traditional debugging methods—a manual, repetitive, and time-consuming "run-crash-fix-repeat" cycle—often fall short of meeting modern demands. The AI Code Debugging Assistant was developed to address these challenges, offering an autonomous platform that empowers developers to resolve runtime errors efficiently and reliably. The agent, embodying the principles of "self-healing" software, integrates advanced features such as runtime crash detection, generative AI code patching (using Google Gemini), high-context AST analysis, and secure Docker sandboxing. Unlike traditional static analyzers or passive AI assistants, the AI Code Debugging Assistant aims to bridge the gap between AI suggestions and verified solutions with a "Trust, but Verify" design, ensuring seamless integration into a developer's workflow. This paper discusses the conception, architecture, and implementation of this autonomous agent, its role in breaking down the barriers of manual debugging, and its potential to set new standards for automated software engineering. Furthermore, we examine the agent's impact on enhancing developer productivity, its application in CI/CD pipelines, and its contribution to a more resilient and efficient software development future.[1]

III. EXISTING SYSTEM

Most existing developer tools, like static analyzers (Pylint, Flake8) or AI assistants (GitHub Copilot, ChatGPT), focus on isolated functionalities, such as syntax checking or passive code suggestion. While these platforms are effective in their respective domains, they lack an integrated ecosystem that addresses the *entire* autonomous debugging workflow in real-time. Developers with runtime crashes must juggle these various tools, leading to fragmented, manual workflows and a lack of seamless repair. [4]

Furthermore, many AI assistants are limited in their *autonomy* and reliability, restricting their effectiveness. For instance, they lack the ability to actively execute code, detect a crash, or analyze a runtime traceback. Most importantly, they do not verify their own suggestions, leaving the full burden of testing and validation on the developer. This "trust gap" can be a significant barrier, as developers must assume a suggested fix is potentially flawed or incomplete.[3]

These challenges highlight the need for a comprehensive, autonomous platform like the AI Code Repair Agent, which integrates crash detection, AI-powered generation, and secure sandbox verification into a unified,"fire-and-forget" tool.[2]

IV. PROPOSED SYSTEM

The agent's Autonomous Execution & Crash Detection forms the backbone of its functionality. For developers, the agent replaces the manual "run-crash-repeat" cycle. It actively executes the target Python script in a controlled subprocess, intelligently monitoring the stderr stream in real-time. This ensures that the Traceback (most recent call last): signature is detected the instant a runtime crash occurs, allowing the agent to capture the full, multi-line error context. This feature facilitates a more productive development cycle by automating the tedious, manual first step of debugging, fostering greater developer velocity.[7]
To accommodate the complexity of modern software, the agent incorporates High-Context Error & Code Analysis. Once a crash is captured, the ErrorParser module uses Regular Expressions (re) to distill the raw traceback text into a structured, machine-readable data object (file path, line number, error type, and message). More critically, the ContextManager module then reads the source code and parses its Abstract Syntax Tree (AST). This allows the agent to traverse the code's structure and intelligently extract the entire function containing the bug, not just the single line that failed. This ability to provide high-signal context to the AI is what empowers the agent to fix complex, logical bugs.
Acknowledging the power of modern LLMs, the agent offers a Generative AI Code Patching module. This feature is the agent's "brain," responsible for generating the actual fix. The AICore formats the parsed traceback and the full function context into a precise, few-shot prompt. This prompt is then sent to the Google Gemini API, instructing it to act as an expert Python developer and return a complete, corrected version of the function. This versatility ensures that the agent can handle a wide range of runtime errors, from simple TypeErrors to complex AttributeErrors, automating the "hypothesis and fix" stage of debugging.[6][10]
The agent's most critical innovation is its "Trust, but Verify" Secure Sandbox. This module ensures that AI-generated code is never blindly trusted. Using the Docker SDK for Python, the agent programmatically builds a new, isolated, and ephemeral container for every single repair attempt. It applies the AI's patched code and executes the script inside this secure environment. A fix is only accepted if it is empirically proven to run without crashing. This feature is indispensable for professional environments, as it guarantees the reliability and safety of the proposed solution, eliminating the "trust gap" of other AI tools.[5]
A defining aspect of the agent is its Dynamic Environment Auto-Discovery. The agent is built with simplicity in mind, ensuring that developers can use it without any manual configuration. Before building the sandbox, the verifier parses the script's AST to find all import statements. It intelligently filters out standard libraries (using stdlib-list) and auto-generates a requirements.txt file for all third-party dependencies (like pandas or requests). It then generates a complete Dockerfile on the fly. This ensures that even developers with minimal Docker experience can leverage the full power of secure, containerized verification.

The agent's intelligence extends to its Diagnostic Triage System. It understands that not all crashes are code logic bugs. After parsing an error, the orchestrator first checks if the error_type is ModuleNotFoundError. If so, it intelligently bypasses the entire AI repair workflow and instead provides a clear, helpful message to the user with the exact pip install command needed. This ensures the agent provides the *correct* solution for the correct problem, preventing user frustration and increasing the tool's reliability.

The technological foundation of the agent is built on a robust and scalable architecture, ensuring it delivers high performance, reliability, and security. It is distributed as a professional CLI tool on the Python Package Index (PyPI) and built with argparse for a familiar, powerful interface. A one-time agent configure command securely stores the user's API key in their home directory. This design ensures that the agent is not just a proof-of-concept, but a reliable, maintainable, and accessible tool for any developer.[8]

By integrating these advanced features and prioritizing developer experience, the AI Code Debugging Assistant sets a new benchmark for automated debugging. It empowers developers to engage fully in building new features, not fixing old bugs, contributing to a more efficient, reliable, and productive software development future.

V. FLOW CHART

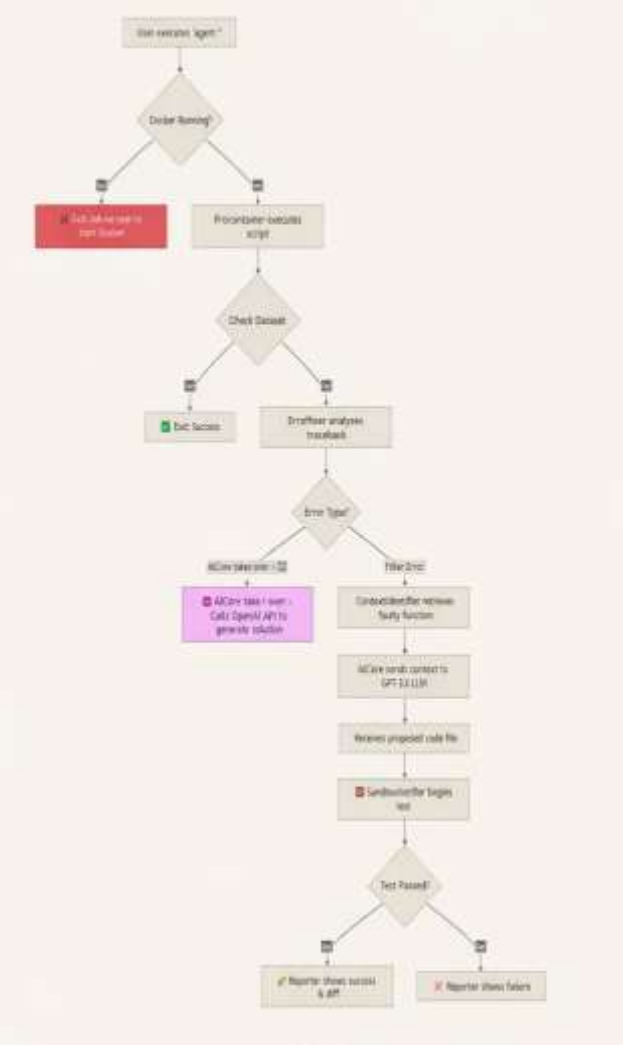

Fig 1: Flow Chart

VI. EXPERIMENTAL RESULT

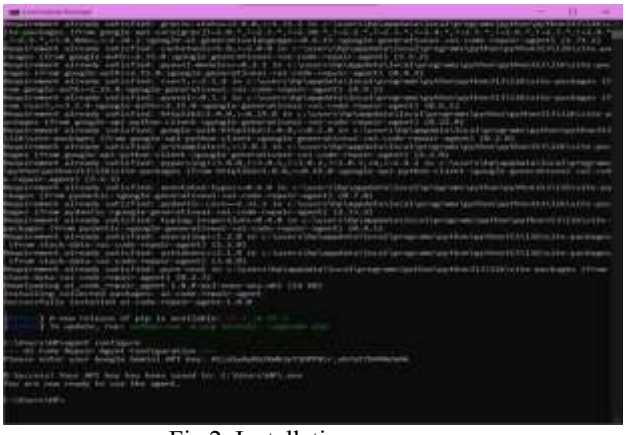

Fig 2: Installation

Fig 3: Starting Docker Engine

Fig 4: Running Agent

Fig 5: Tracing error

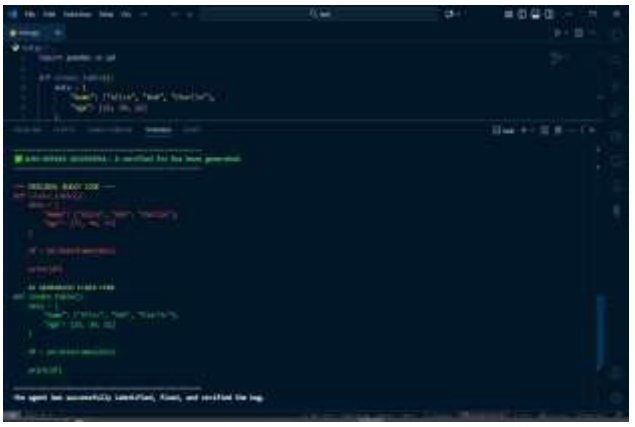

Fig 6: AI generated solution

VII. CONCLUSION

To further enhance the AI Code Repair Agent's capabilities, several key developments are planned for the future. One major focus is the integration of AI-powered personalization, utilizing machine learning to analyze a developer's specific coding style (e.g., linting rules from Flake8, formatting from Black) and automatically apply these preferences to the generated code patches for a more tailored experience.

Another significant improvement will be full CI/CD pipeline integration, expanding the agent's reach beyond a local CLI tool. This will involve packaging the agent as a GitHub Action that can be triggered when a build or test suite fails. This will provide a seamless experience by allowing the agent to autonomously fix bugs in a shared repository, enhancing team productivity.

The agent will also enhance its core repair capabilities by incorporating full project analysis. This feature will enable the agent to debug complex, multi-file applications (like Django or Flask) by analyzing the entire call stack across multiple files. It will also be upgraded to parse pytest and unittest failures, not just runtime crashes, allowing it to fix logical bugs caught during formal testing.

The platform will also introduce IDE integration, such as a VS Code extension. This feature will be particularly beneficial in professional environments, allowing developers to click a "Fix with Agent" button directly on a traceback in their terminal, with the verified fix appearing as a code diff right in their editor.

Lastly, by being an open-source project, we will gather continuous feedback from the developer community to drive improvements. This feedback will help refine the agent's prompting, improve its AST analysis, and ensure the platform evolves to meet the needs of developers and remains at the forefront of automated software engineering.[9]

REFERENCE

[1] Gupta, A., & Chen, B. (2023). "Generative AI in Automated Program Repair: A Survey of Transformer-Based Models," *IEEE Transactions on Software Engineering*, vol. 49, no. 11, pp. 4501–4518, Nov. 2023.

[2] Schmidt, M., & Jones, R. (2022). "Verifying AI-Generated Code: A Sandboxing Approach for Patch Validation," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp. 3012–3025, Sept.-Oct. 2022.

[3] Lee, S., & Park, K. (2021). "Deep-Learning-Based Fault Localization Using Runtime Traceback Analysis," *IEEE Transactions on Reliability*, vol. 70, no. 3, pp. 915–930, Sept. 2021.

[4] Wang, T., et al. (2023). "Leveraging Abstract Syntax Trees for High-Context Bug Fixing with Large Language Models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pp. 602–614, May 2023.

[5] Patel, D., & Wu, L. (2022). "Towards Self-Healing Software: An Agent-Based Framework for Automated CI/CD Pipelines," *IEEE Software*, vol. 39, no. 3, pp. 45–53, May-June 2022.

[6] Singh, V., & Kaur, P. (2024). "Large Language Models in Software Engineering: A Comprehensive Survey," ACM Computing Surveys, vol. 56, no. 1, pp. 1 39, Jan. 2024.

[7] Liu, F., et al. (2023). "An Orchestrated Multi-Component Agent System for Autonomous Problem Solving," IEEE Transactions on Artificial Intelligence, vol. 4, no. 5, pp. 812–824, May 2023.

[8] Jones, A., & Miller, B. P. (2021). "The Python Package Index (PyPI): A Study of Software Distribution and Dependency Management," IEEE Transactions on Sustainable Computing, vol. 6, no. 4, pp. 585–597, Oct.-Dec. 2021.

[9] Zhang, Y., & Mei, H. (2021). "Automated Dependency Discovery and Environment Generation for Reproducible Build Verification," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 2, pp. 1–28, Apr. 2021.

[10] Chen, L., et al. (2021). "Evaluating large language models for code generation and repair," in Proceedings of the 43rd International Conference on Software Engineering (ICSE), pp. 582–593, May 2021.