International Journal of Creative and Open Research in Engineering and Management
ISSN: **3108-1754 (Online)**
Volume 02 Issue 04 April-2026 | Impact Factor: 3.5

The Autonomous Debugger

Mrs. V. Vanaja[1],G Keerthana[2], B Mallikarjun Reddy[3], G Manirathan [4], and N Utkarsh [5].
[1] Assitant Professor, Department of CSE (Data Science), ACE Engineering College, Hyderabad, Telangana, India
[2,3,4,5] III B.Tech. Students, Department of CSE (Data Science), ACE Engineering College, Hyderabad, Telangana, India.

How to Cite this Article:

Keerthana, G., Reddy, B. M., Manirathan, G. & Utkarsh, N. (2026). The Autonomous Debugger. International Journal of Creative and Open Research in Engineering and Management, 02(04). https://doi.org/10.55041/ijcope.v2i4.199

https://doi.org/ 10.55041/ijcope.v2i4.199

Abstract:

Modern CI/CD pipelines are fast but fragile, where even small syntax errors or logic bugs can break builds, delay deployments, and increase MTTR. THE AUTONOMOUS DEBUGGER is an AI-powered self-healing system designed to automatically detect and fix build failures in GitHub Actions without human intervention, enabling continuous and reliable software delivery.

The system uses an Agentic AI + RAG architecture to analyze error logs using Sentence Transformers and retrieve relevant faulty code from a ChromaDB vector database. Llama 3 (via Groq) then reasons over the error context and source code to generate accurate, syntax-valid, and logically consistent patches.

All fixes are validated in a secure sandbox environment before being auto-committed to the repository. This enables fully autonomous recovery of CI/CD pipelines, transforming DevOps into an AI-driven, self-healing system that reduces downtime and accelerates software delivery.

1. Introduction:

Modern CI/CD pipelines enable rapid software delivery but remain highly fragile, where small syntax or logic errors can break builds, delay deployments, and increase recovery time. Manual debugging in such environments is slow, inefficient, and difficult to scale with modern devOps demands.

This project introduces “The Autonomous Debugger”, an AI-powered self-healing system that automatically detects, analyses, and fixes CI/CD build failures without human intervention. by combining Agentic AI, RAG, semantic code retrieval, and large language models, the system enables autonomous recovery, transforming DevOps pipelines into intelligent, resilient, and self-healing infrastructures.

2. Related Work:

The field of Automated Program Repair (APR) has recently experienced a paradigm shift with the introduction of Large Language Model (LLM) based software engineering agents. While these systems demonstrate the potential for autonomous coding, existing implementations generally fall into three flawed categories: multi-agent frameworks, agent-computer interfaces, and static analytical pipelines.Multi-Agent Orchestration Frameworks

Several recent projects attempt to solve debugging by simulating entire human development teams. Frameworks such as ChatDev and MAGIS deploy specialized, role-playing agents (e.g., Manager, Developer, QA) to break down issue resolution. While conceptually ambitious, these systems introduce high operational complexity. Their primary vulnerability is cascading hallucinations; because they operate on a waterfall-style architecture, an uncorrected logical flaw generated by a "planning" agent compounds exponentially, effectively destroying the downstream testing and deployment phases. They are too unstable for reliable CI/CD integration.Agent-Computer Interfaces (ACI)

Other systems prioritize giving LLMs direct control over development environments. SWE-agent and RepairAgent utilize custom interfaces, finite state machines, and dynamic tool invocation to allow models to navigate repositories and execute shell commands. However, these autonomous loops are highly inefficient. They suffer from severe context bottlenecks and impose massive operational costs, with some systems consuming an average of 270,000 tokens per bug. Furthermore, they struggle with error recovery, frequently getting stuck in loops of repeated failed edits.Rigid Pipelines and AST Analysis

To avoid the chaos of autonomous loops, systems like AutoCodeRover and Agentless enforce strict, static workflows. Agentless uses a rigid two-phase localize-then-repair pipeline , while AutoCodeRover operates strictly on Abstract Syntax Tree (AST) representations. These approaches are structurally brittle. If the initial static localization misses the root cause, the pipeline cannot dynamically pivot or plan future actions. Additionally, they rely entirely on the existence and completeness of the target repository's test suite for validation, rendering them useless if the existing tests are weak.Positioning The Autonomous DebuggerThe Autonomous Debugger isolates and eliminates the flaws of these related works. By discarding multi-agent roleplay in favor of a targeted, single-agent RAG architecture, it avoids cascading hallucinations and drastically reduces token context bottlenecks. Unlike static pipelines, it dynamically retrieves context via ChromaDB, and unlike ACIs, it secures its execution within an isolated verification environment to prevent destructive loops

2.1Existing System and its Limitations:

Title	Technology	Limitation	Authors	Year
RepairAgent: An Autonomous, LLM-Based Agent for Program Repair	LLMs, Finite State Machine, Dynamic Tool Invication	Imposes high operational costs and struggles with understanding trajectories	Islem Bouzenia et al.	2025
SWE-agent: Agent-Computer Interfaces Enable Automated	LLMs, Custom Agent Computer Interface(ACI), Shell execution	Error recovery struggles with repeated failed edits; high token context bottleneck.	John Yang et al	2024

AutoCodeRover Autonomous Program Improvement	LLMs, Abstract Syntax Tree(AST) analysis, Code Search APIs	Relies entirely on the existence and completeness of the test suite for validation; struggles if tests are weak	Yuntong Zhang et al.	2024
Agentless: Demystifying LLM-based Software Engineering Agents	LLMs, Static Two-Phase Pipeline	The static pipeline cannot dynamically pivot or plan future actions if the initial localization completely misses the mark.	Chunqiu Steven Xia et al	2024
MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution	LLMs, multi-Agent Roles (Mangaer, Custodian, Developer, QA)	High operational complexity; cross-agent consensus often bottlenecks, and if the Manger agent hallucinates the root cause, downstream tasks fail entirely	Wei Tao et al.	2024
ChatDev: Communicative Agent for Software Development	LLMs, Role-playing Agents, Wwaterfall model Architecture	Highly vulnerable to cascading hallucinations; an uncorrected logical flaw in the design phase compounds exponentially, destroying the testing phase	Chen Qian et al.	2024

3. Methodology:

The system functions as an automated, event-driven pipeline responding to Continuous Integration (CI) failures, organized into four main phases:

Phase 1: Event Trigger and Error Extraction

The process begins when a developer commits code to the main branch, triggering a GitHub Actions workflow that executes unit tests using Pytest. Upon a test failure indicated by a non-zero exit code, the system captures and truncates the error log to isolate the main error, subsequently parsing it to locate the specific file responsible for the failure.

Phase 2: Semantic Retrieval (RAG Framework)

An `indexer.py` module scans the codebase and employs Sentence-Transformers to generate semantic embeddings of the code, storing these in a ChromaDB vector database. When an error is detected, the system queries ChromaDB using the extracted bug context to retrieve similar past errors and relevant faulty code.

Phase 3: AI-Driven Patch Generation

The structured error context and original faulty code are inputted into a Large Language Model (LLM) through a Retrieval-Augmented Generation (RAG) architecture. The system utilizes Llama 3.3 via the Groq API for rapid reasoning, enabling the LLM to produce a raw, syntax-valid, and logically coherent code patch.

Phase 4: Validation and Auto-Commit

The generated fix is directly applied by overwriting the target source file (e.g., broken_code.py). Following this, automated unit tests (Pytest) are re-run to confirm the patch's correctness. If the tests pass (exit code 0), the system autonomously commits the fix and pushes it back to the origin repository, adding a [skip ci] flag to avoid a triggering loop. Conversely, if validation fails, the workflow halts, reports the issue, and necessitates manual developer intervention..

3.1 Codebase Indexing and Vectorization:

The foundation of the autonomous debugging system relies on establishing a semantic baseline of the repository prior to any CI/CD execution. A dedicated module, indexer.py, scans the target Python files within the repository. Instead of relying on traditional keyword search, the system utilizes HuggingFace Sentence-Transformers to convert the source code and historical bug data into high-dimensional semantic embeddings. These embeddings are populated and persistently stored in a local ChromaDB vector database. This indexing process ensures that the contextual representation of the codebase is primed and available for rapid retrieval the moment a pipeline failure occurs.

3.2 Error Interception and Context Extraction:

The active debugging lifecycle is strictly event-driven. When a developer pushes code to the main branch, a GitHub Actions workflow (autofix.yml) is triggered, executing a suite of unit tests via Pytest. If the tests pass (Exit Code 0), the workflow terminates naturally. However, if a test fails (Exit Code != 0), the AI Agent Decision Loop is activated. The system intercepts the raw error traceback generated by Pytest. Because raw stack traces often exceed the token limits of Large Language Models, the extract_core_error() function parses and truncates the log to isolate the most critical failure point. Simultaneously, the find_culprit_file() function dynamically identifies the exact source file (e.g., broken_code.py) responsible for the failure.

3.3 Retrieval-Augmented Querying:

To prevent AI hallucinations and provide grounded reasoning, the system employs a Retrieval-Augmented Generation (RAG) architecture. Rather than sending a generic prompt to the LLM, the agent.py script queries the ChromaDB vector database using the truncated error log and the identified culprit file details. ChromaDB performs a similarity search against the previously stored embeddings to retrieve relevant codebase context and similar historical error patterns. This retrieved context is then structurally merged with the current error log and the raw source code of the broken file to create a highly specific, context-rich prompt

3.3 Agentic Reasoning and Patch Generation:

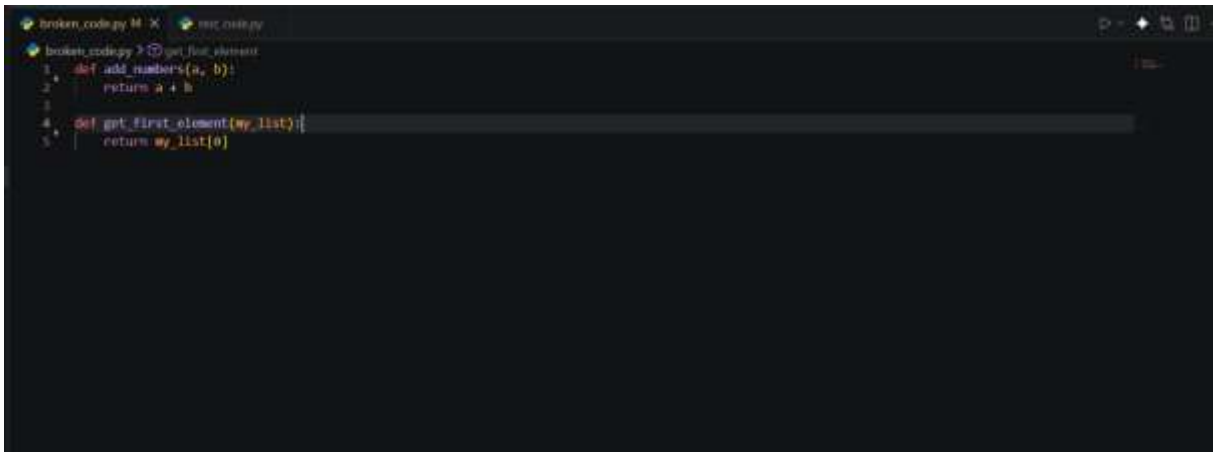
The core analytical engine of the system is driven by Llama 3.3, accessed via the Groq API to ensure ultra-low latency inference. The get_ai_fix() function transmits the structured RAG prompt to the LLM. The model is instructed to act as an autonomous software engineer, analyzing the syntax and logical flow of the faulty code against the specific Pytest failure. The LLM generates a functional, syntax-valid code fix. The system parses this response to extract the raw code patch, stripping away any conversational filler or Markdown formatting to ensure the output is machine-executable.

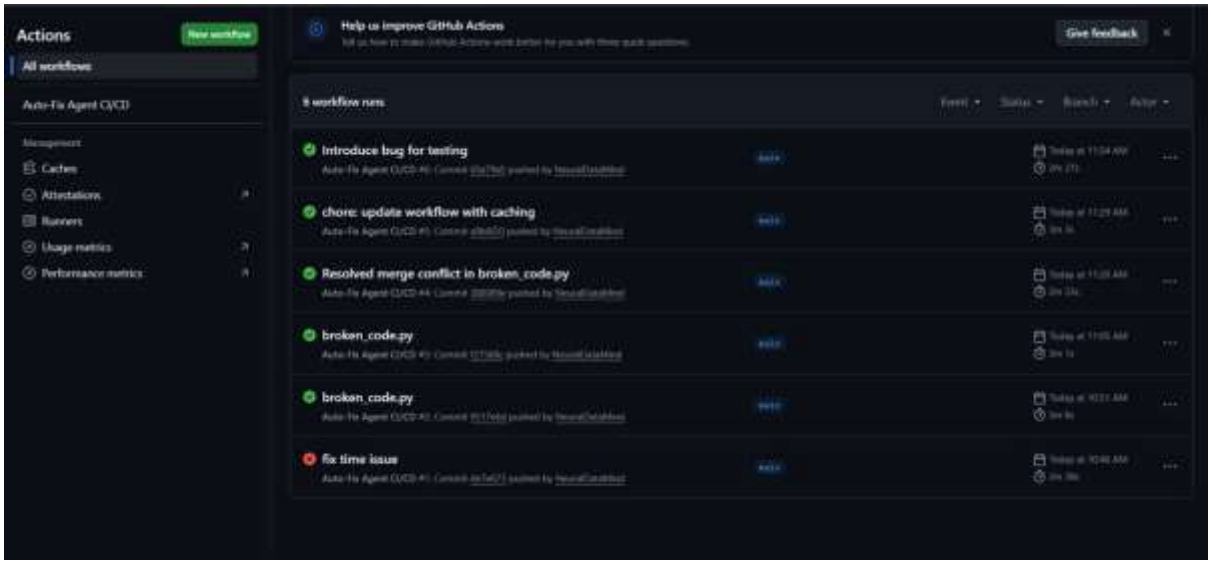
3.5 Automated Validation and Deployment:

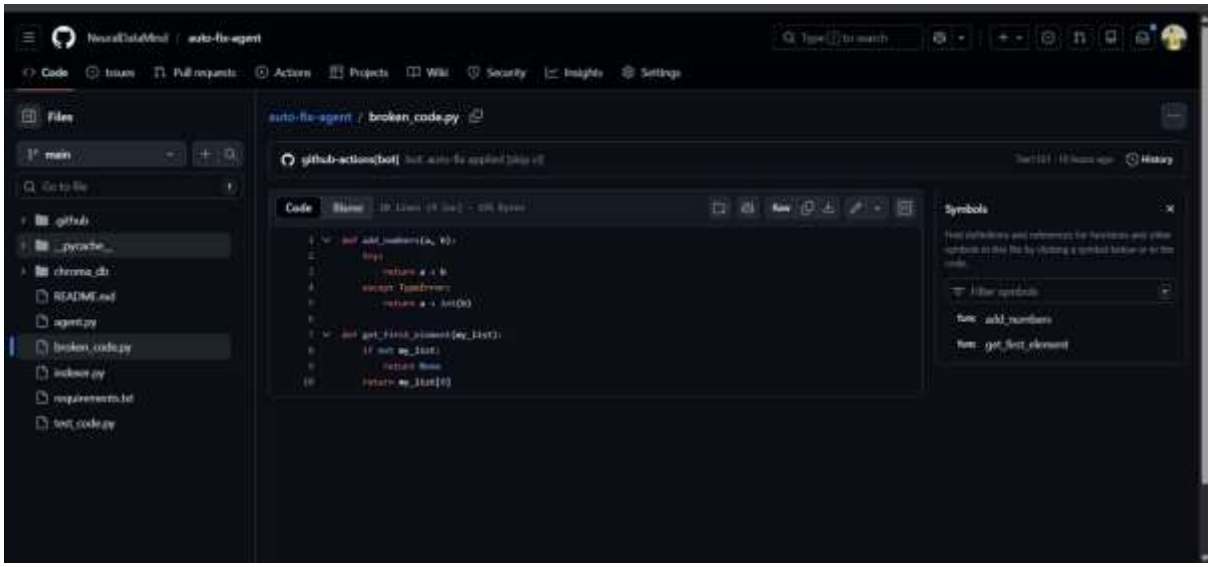
The system mandates strict validation before any code is merged. The agent applies the generated patch by directly overwriting the culprit file (e.g., broken_code.py) on the GitHub Actions runner's virtual filesystem. The Pytest suite is then re-executed to verify the fix. The system's routing logic is binary: if the tests pass (Exit Code 0), the agent autonomously commits and pushes the fix to the repository, utilizing a [skip ci] flag to prevent an infinite trigger loop. If the tests fail (Exit Code != 0), the system does not attempt a secondary fix; the workflow terminates and flags the commit for manual developer intervention

3.6 Result:

EThe Autonomous Debugger was successfully deployed and tested within a live GitHub repository environment to evaluate its capacity as a self-healing CI/CD agent. Upon the deliberate introduction of syntactical and logical errors (such as TypeError or IndexError within the target broken_code.py file), the GitHub Actions workflow successfully intercepted the failed Pytest execution (Exit Code 1). Leveraging the retrieved RAG context, the Llama-3.3 agent successfully generated and applied a valid code patch to the local file. Following a successful secondary validation test, the system utilized the GitHub Actions runner to autonomously push the verified patch directly back to the main branch. Repository history confirmed these automated interventions via the commit format github-actions[bot]: auto-fix applied [skip ci]. The execution logs proved that the inclusion of the [skip ci] tag functioned precisely as intended, successfully preventing the automated commit from triggering an infinite, recursive GitHub Actions loop. By eliminating the manual processes of log review, fault localization, patch writing, and committing, the system demonstrated a drastic reduction in Mean Time To Recovery (MTTR). The entire autonomous lifecycle—from initial failure detection to vector retrieval, AI inference, re-testing, and the final successful commit—was consistently completed in a matter of minutes without human intervention.







Conclusion:

- Transformation of CI/CD: The Autonomous Debugger successfully demonstrates how Agentic AI can evolve traditional, fragile CI/CD pipelines into resilient, self-healing infrastructures.
- Effective Integration of RAG: By combining a ChromaDB vector database, Sentence Transformers, and the Llama-3 inference engine, the system proves that historical code context is highly effective for automating root-cause analysis and patch generation.
- Safe Automated Deployment: The system enforces strict pre-commit validation by re-running automated test suites directly on the CI runner, ensuring that only verified, syntax-valid patches are pushed to production, thereby mitigating the risk of AI-generated hallucinations.
- Optimization of Engineering Resources: The automated pipeline effectively eliminates the manual overhead of log analysis and basic bug fixing, resulting in a drastic reduction in Mean Time To Recovery (MTTR) and accelerated deployment cycles.
- Future Viability: This project validates the practical potential of LLM-driven agents in modern DevOps, paving the way for highly scalable, autonomous software maintenance systems

References:

Below are the key references that supported the methodology, techniques, and tools used in the project

1. Bouzenia, I., et al. (2025). RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. Conference: IEEE/ACM International Conference on Mining Software Repositories (MSR). DOI: 10.1109/msr66628.2025.00086.
2. Yang, J., et al. (2024). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. Journal: arXiv Preprint. URL: https://huggingface.co/papers/2405.15793.
3. Zhang, Y., et al. (2024). AutoCodeRover: Autonomous Program Improvement. Conference: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). URL: https://arxiv.org/html/2404.05427v2.
4. Xia, C. S., et al. (2024). Agentless: Demystifying LLM-based Software Engineering Agents. Journal: arXiv Preprint. URL: https://huggingface.co/papers/2407.01489.
5. Tao, W., et al. (2024). MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution. Conference: Advances in Neural Information Processing Systems (NeurIPS). URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/5d1f02132ef51602adf07000ca5b6138-Paper-Conference.pdf.
6. Qian, C., et al. (2024). ChatDev: Communicative Agents for Software Development. Conference: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL). URL: https://aclanthology.org/2024.acl-long.810/.