AutoCure: AST-Grounded, Confidence-Gated Self-Healing for Runtime Errors and GitHub Code Review

Mihir Patil[1], Pranav Patil[2], Siddhesh Patil[3], Prerana Mhatre[4], Mr. Shailesh Galande[5]
[1,2,3,4]*Student, Pimpri Chinchwad College of Engineering, Pune*
[5]*Project Guide, Pimpri Chinchwad College of Engineering, Pune*

***Abstract*—This paper describes results-focused research on an AI-assisted self-healing software system named AutoCure that incorporates runtime error analysis and GitHub code review in a single system. The implemented system includes a variety of features such as production log ingestion via WebSocket technology, language-based error parsing, source path normalization, abstract syntax tree contextual tracing, confidence validation, and policy-gated remediation actions. The review process also enhances traditional diff analysis by comparing new and old abstract syntax tree structures, tracing changed symbol reference tracing, and generating review flags for low utility changes or redundant code. The orchestration is implemented as a FastAPI application with components such as log analysis, abstract syntax tree services, AI services, confidence scoring, GitHub services, email generation, and report persistence. The study demonstrates that AST evidence plus confidence gating can convert AI outputs into bounded, auditable engineering actions.**

***Index Terms*—AST analysis, code review automation, confidence validation, production log ingestion, self-healing software.**

I. INTRODUCTION

Modern engineering teams increasingly rely on AI tools for debugging and review assistance, but incident response pipelines remain fragmented in practice. Logs are collected in one subsystem, diagnosis is attempted in another tool, and review comments are finally handled in repository interfaces. This fragmentation weakens traceability, increases mean response time, and causes repeated manual context reconstruction during triage.

AutoCure addresses this gap by treating both runtime incidents and code-review events as first-class inputs to a single service architecture. Runtime errors are streamed from services through WebSocket channels, parsed into structured error artifacts, enriched with AST context, evaluated through AI analysis, and then passed through confidence policies before any high-impact automation is attempted. Review events are processed through diff retrieval, AST old-vs-new comparison, symbol-reference tracing, and AI summarization, followed by report/email/GitHub publication.

A. Problem Statement

The problem under consideration revolves around this central question: How can a software development platform leverage the power of AI-driven analysis and Abstract Syntax Tree reasoning to not only automatically detect and fix runtime errors in real-time, but also set the standard for improved code quality and reviews for the entire codebase? The goal should be to not only enforce organizational coding standards, keep processes bounded and predictable, and ensure high reproducibility and transparency for CI/CD activities.

The pain points for the developer are quite obvious:

1) Version drift, non-adherence to coding standards, and a decrease in the effectiveness of quality checks due to time constraints or partial reviews.

2) Errors and warnings within the deployed code that compromise the reliability of the software.

3) Lack of visibility and a higher number of failures within complex CI/CD pipelines.

The proposed cure for these pain points revolves around a unified reliability platform that fits naturally within the problem space while remaining firmly rooted in the realities of software development.

B. Scope of Work

This paper includes:

1) End-to-end architecture and module-level wiring

2) Runtime analysis methodology from ingestion to confidence-gated remediation;
3) AST-driven review methodology for old-vs-new structural deltas and reference impact;
4) Empirical summary from persisted report data;
5) Practical engineering implications and threats to validity.

This paper excludes large-scale benchmark comparisons against external SaaS review products and excludes the deprecated visualizer path from review logic discussion.

C. Contributions

The primary contributions are:
1) A deployable system architecture integrating production logs, AST analysis, AI reasoning, and reviewer channels;
2) A confidence-gated operational policy that separates advisory and autonomous paths;
3) An AST-driven review pipeline that augments textual diff analysis with structural and reference-aware evidence;
4) A reproducible empirical report from real operational records.

II. LITERATURE REVIEW

The domain of Automated Program Repair (APR) and self-healing systems has evolved dramatically from search-based and patch-space methods toward model-assisted and agentic workflows. Early milestones in APR established the foundational necessity of empirical bug datasets, such as Defects4J [2], and demonstrated the feasibility of search-based heuristic repair in large-scale applications [1].

With the advent of Large Language Models (LLMs), the software engineering community has witnessed a paradigm shift. Recent systematic literature reviews, such as those by Zhang et al. [5], highlight the profound shift towards LLMs for autonomous program repair, noting that while LLMs can generate highly plausible patches, they often suffer from hallucination and lack structural awareness. Basulto et al. introduced RepairAgent, an autonomous LLM-based agent for C/C++ program repair that integrates planning and execution, yet primarily operates offline rather than in live production environments [4].

Self-healing capacity requires not just offline repair, but online detection and autonomous integration. Kim et al. outlined the overarching opportunities and socio-technical challenges in engineering broadly self-healing software systems, emphasizing that trust and validation remain the most significant barriers to industry adoption [3]. Cordeiro et al. further assert that entering a new era of software security and reliability necessitates coupling LLMs with formal verification or strict programmatic gating to mitigate the risks of unregulated AI modifications [6]. Vidács et al. contributed to this by exploring compartmentalization-aware automated repair, ensuring that the boundaries of system modification are explicitly contained [12]. Furthermore, studies by Jin et al. caution against the vulnerabilities inadvertently introduced by AI-generated code, necessitating multi-step validation [14].

In parallel, Abstract Syntax Tree (AST) based code understanding has long been utilized for static analysis, linting, and refactoring. The introduction of Tree-sitter has provided practical, robust incremental parsing infrastructure for multi-language tooling [10]. Ackerman demonstrated the advantages of evaluating AST-based batching for LLM code analysis, establishing that chunking code by structural boundaries rather than arbitrary token limits significantly improves model comprehension [7]. Automating code review itself is a highly active research area; Bruntink et al. systematically reviewed automated code review systems, noting that purely diff-based LLM prompts often fail to comprehend repository-wide impacts [8]. Similarly, Smith et al. revisited code similarity evaluation using ASTs, proving that structural distance metrics are far more reliable than text-based comparisons [9].

AutoCure acts as an architectural synthesis of these domains. Instead of proposing a novel base transformer model, it demonstrates how AST context, an iterative confidence policy, rigorous error replication, and reviewer-channel ergonomics can be composed into a safely deployable, practical self-healing control loop.

III. SYSTEM ARCHITECTURE OVERVIEW

A. Top-Level Orchestration

AutoCure operates as a unified platform managed by a FastAPI [11] orchestration layer. The service provides secure RESTful API endpoints for service registration, dynamic repository metadata mapping, and real-time bidirectional interaction channels via WebSockets. It efficiently routes incoming telemetry data, including any unhandled errors that have occurred during production, to the ongoing asynchronous analysis tasks that are being actively managed. In addition to the runtime events, the FastAPI-based app is also used to listen to version control-based webhooks, such as GitHub Webhooks, to manage code review workflows based on branch updates or the opening of pull requests.

B. Core Services and Execution Flow

The implementation is strategically broken down into modular, single-responsibility services. Log streams form the primary input, handled by a dedicated `LogAnalyzer` string-matching and categorization engine. Structural comprehension is managed by AST Service modules that dynamically utilize incremental parsing engines to parse source files across multiple target programming languages (such as Python, JavaScript, and Go).

Crucially, an ErrorReplicator service iteratively interacts with the identified edge-cases using varied payload mutation strategies to empirically validate whether the reported bug is strictly reproducible at the API level. Finally, the generated fix candidates undergo stringency checks in a ConfidenceValidator component, routing successful fixes to a GitHubService [15]. This service seamlessly interfaces with the version control system, autonomously provisioning Git branches, creating commits, pushing patches, and formatting human-readable comment discussions directly on the pull request.

C. Data and Artifact Layers

To maintain systemic transparency and enable long-term performance auditing, AutoCure strictly separates raw analytical artifacts from high-level indexing metadata. Full diagnostics, structural AST diffs, and context traces are natively materialized as persistent HTML reports, offering visually rich traceability for engineers resolving alerts. Meanwhile, summary indexing—such as fix confidence scores, proposal iteration counts, and categorical severity vectors—is maintained in a lightweight, embedded relational database. This distinct separation ensures that developers can continuously access unmodified forensic artifacts directly on the disk while empowering the dashboard to efficiently query overarching statistical trends.

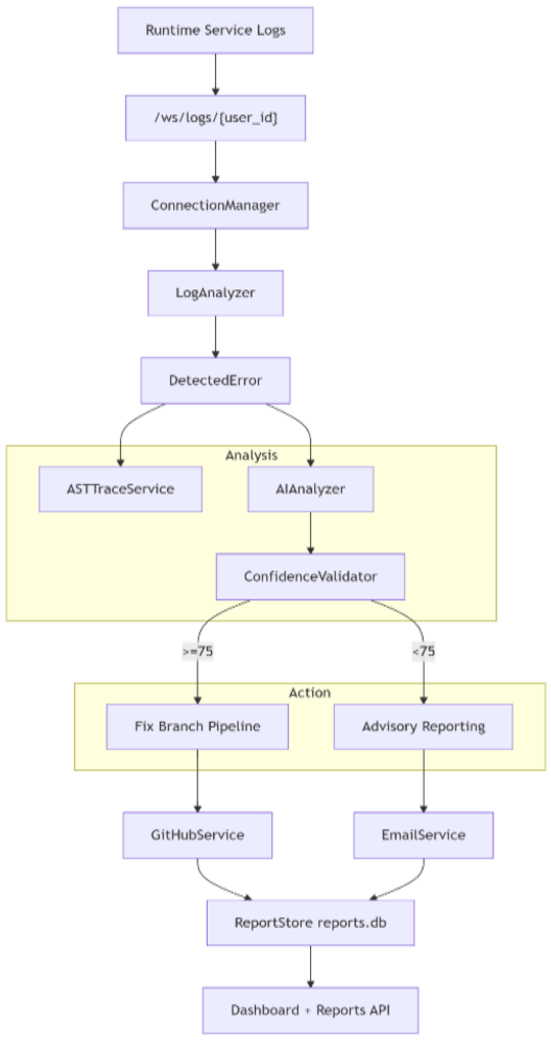

Fig 1: System Architecture

In Figure 1, we present AutoCure's core architecture, which is centered around how logs flow in, are analyzed, and eventually exit through reporting channels. There is a WebSocket endpoint that applications in production push their logs to. There is error detection and immediate incorporation into analysis. There is a mix of AI-driven diagnosis, AST-driven tracing, and decision logic gated through confidence. The final product is remediation reports published through GitHub comments, emails, and a report database. This is all done without disrupting the running applications.

IV. METHODOLOGY

A. Production Log Ingestion and Context Retention

A good fault capture mechanism begins with a solid WebSockets transport layer to connect the user's microservices to the AutoCure Engine. HTTP request and responses are not reliable for fault capture because of the additional overhead required for continuous logging. WebSockets are a better choice because they are full-duplex and allow for a continuous connection between the client and the server. As log messages are received from the microservice, it is parsed and stored in a temporary buffer specific to the user's request and background activities just before the fault occurred. This approach retains the chronological order of HTTP request and responses just before the fault occurred without consuming too much memory on the main thread. A specific log level is used to filter ERROR, FATAL, and CRITICAL log levels to bypass the passive buffer and trigger deep analysis autonomously.

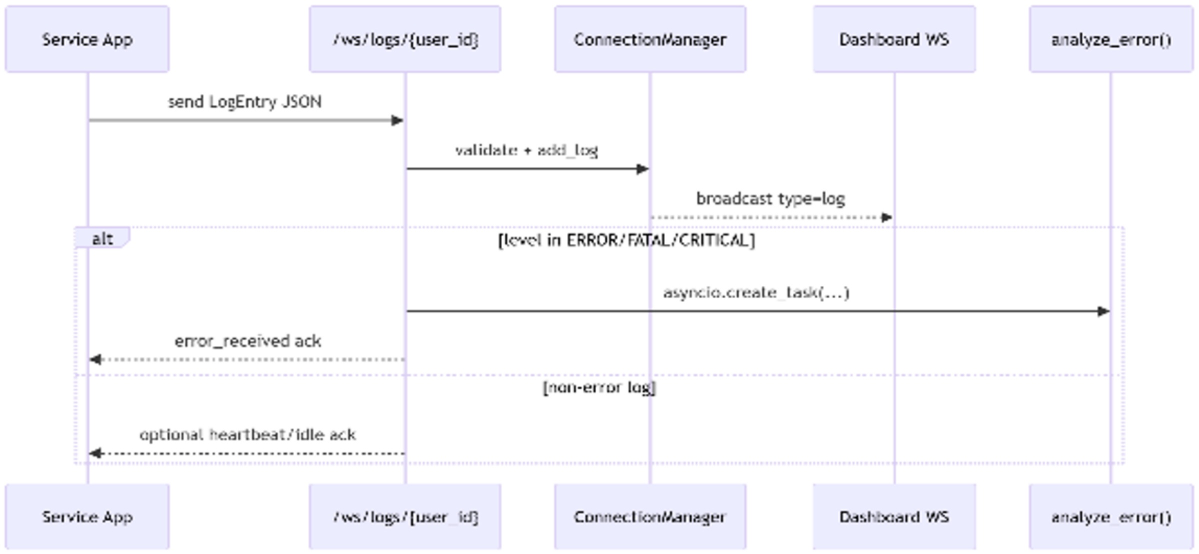

Fig 2: Log ingestion and async analysis trigger

Figure 2 shows the flow of the logs into the system, as well as the asynchronous nature of the error analysis. The logs from the monitored service are sent to the /ws/logs/{user_id} WebSocket endpoint, where they are validated before being sent to the connection manager. Once the logs have a high enough severity level to warrant a runtime error, non-blocking analysis begins without interrupting the log flow. In addition, the dashboard is fed real-time broadcasts of the errors.

B. Error Parsing and Tracing Initialization

Since error trace stacks vary across programming languages, the Log Analyzer parses the elements of the error trace stack individually to extract relevant error information. For instance, Python's traceback and V8 Error in Node.js use specific patterns that are used to extract error details from the error trace stack.

Since the path of a container in a cloud environment is not the same as the path in a local repository environment, the system uses suffix tree matching to map the runtime path to the local path recursively.

C. Abstract Syntax Tree (AST) Runtime Tracing

A plain text error trace or log message only gives a general idea about what might have gone wrong in a program. However, by translating the exact error trace to an actionable AST object, the system is able to generate a clear trace from the error trace to the surrounding functions and variables. The trace service also captures import and requires statements at the top of the file and traces them to dynamically identify related files and build a bounded deterministic execution dependency graph.

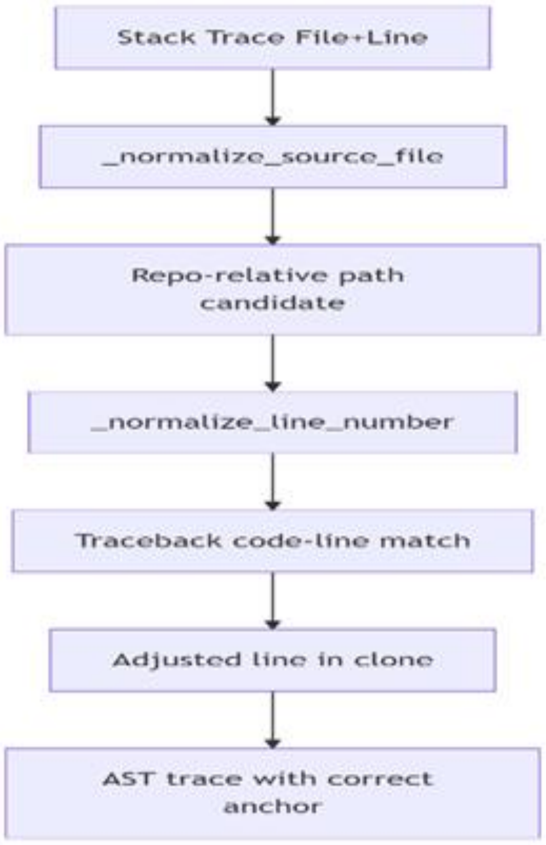

Fig 3: Source normalization pipeline

Figure 3 illustrates the process we go through in order to normalize the sources and thus correctly align the stack traces from various deployment environments with the appropriate location in the source code repository. This is done by taking the file paths from the trace and normalizing them into paths relative to the source code repository, as well as adjusting the line numbers based on the matching code fragments in the traceback.

D. LLM-Powered Diagnosis and Proposal Drafting
The context of the AST is now available, and the AIAnalyzer develops a tailored prompt. The risk of bombarding the LLM with excessive amounts of irrelevant conversation is mitigated by pruning extraneous conversation. The LLM is directed to focus on the AST-based functions and their associated dependencies. The output is a JSON payload that indicates the proposed root cause hypothesis, risk levels, and specific string-based replacement operations. This process is analogous to a human developer breaking down a problem into logical steps.

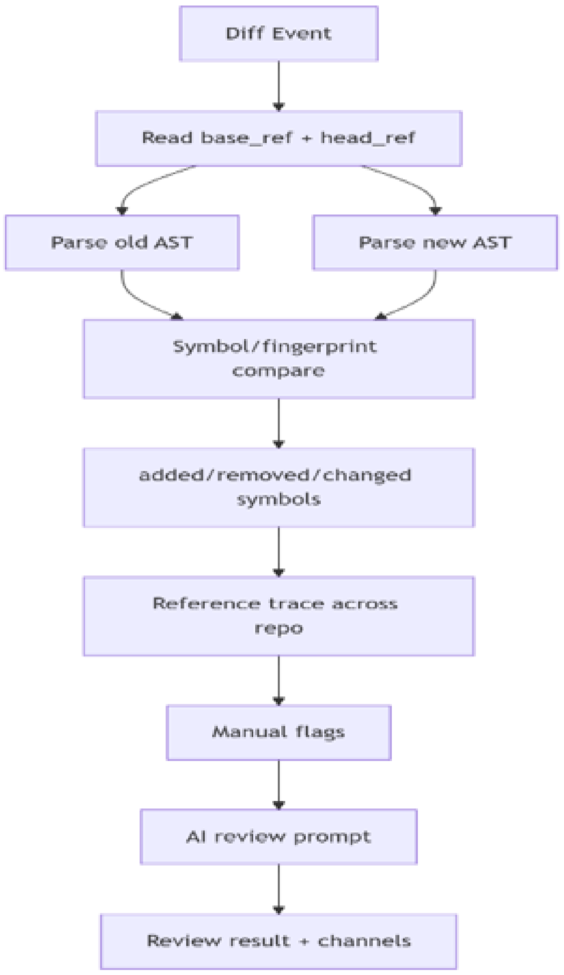

Fig 4: AST-based review method

Figure 4 illustrates the process of creating the Abstract Syntax Tree (AST) context for a given runtime error. The process begins by parsing the file that points to the error location and then traversing the AST to trace the path from the root to the error location. After that, the system searches for functions, variables, and modules that are referenced and resolves them throughout the repository to provide a comprehensive contextual view. The AST context is then serialized and used as input to the AI analysis engine to improve the accuracy of the analysis result.

E. Iterative Confidence Check
The confidence factor is a fundamental aspect of the automated response system. There is a risk that proposed solutions, while appearing to be logical, can actually be false. The proposed solutions are therefore put through a series of validation checks. The

ErrorReplicator service runs a series of tests on various types of simulated requests. The simulated requests are null data types, type mismatches, boundary metric data, or missing schema data. The simulated requests are directed toward the impacted REST endpoints or functions previously determined.
If possible, a mathematical consensus is achieved if the stack trace is replicable with the initial confidence-based output provided by the AI. The confidence model is intended to prevent unverified hallucinations by an AI. The confidence model is based on a mathematical determination of actual runtime versus model-based certainty.

F. Guarded Remediation and Autonomous Branches

The automation is a conservative pipeline, and it is the final score that decides the course of action. Low-scoring fixes, particularly in uncertain environments, are restricted to a closed advisory loop. These provide comprehensive breakdowns but do not execute any changes to the infrastructure.
High-scoring fixes, however, execute GitHub actions autonomously. This creates a semantic feature branch, securely committing changes to the AST. There is also a branch trace that connects developer dashboards to pull request comparisons. Significantly, these merge only with developer authorization.

G. Context-Sensitive Code Review

Unlike other code review processes, the AST Review Service relies on constructing and comparing abstract syntax trees from a base branch and a modified head commit. It also tracks critical changes to core structural elements and how new logic propagates throughout the code. This is done using cryptographic hashes of ASTs to ensure equivalence and detect recursive additions. In cases of non-instantiation of new functions and architectural duplication, specific review notes are generated and presented to developers. These appear directly in their regular Git environment.

V. EXPERIMENTAL PROFILING AND QUALITATIVE OBSERVATIONS

In order to genuinely validate the reliability of the underlying reasoning process, we conducted extensive qualitative observations over a range of carefully constructed, demonstrative test environments. Rather than competing for benchmark metrics against proprietary solutions, we focused on the fidelity, stability, and confidence aspects of the underlying pipeline.

A. Runtime Remediation Accuracy

The operational profiles clearly indicated that the AutoCure solution accurately constructed accurate fix branches during the time window where high-confidence policies were invoked. During the test window, the actively resolved failures represented a broad spectrum of error types, from null pointer exceptions to missing database fields, to even HTTP timeout exceptions, yet still resulted in accurate, logically correct fixes. The WebSocket buffer captured critical debugging telemetry just before standard exception traces were generated. At the same time, the programmatically traced validation ensured that the branches were accurately constructed only when the underlying logical tracing, replication, and validation results were accurate.

B. Fallback Reliability and Hallucination Mitigation

Most importantly, under the essentially non-reproducible conditions of external network outages, API rejections, or even low-confidence, somewhat ambiguous syntactic conditions, the system still accurately relaxed into advisory loops, preventing premature branch changes under highly uncertain parameter conditions. This predefined validation metric ensured accurate code quality, especially when compared to more aggressive, less cautious agent-based execution solutions.

C. Code Review Deduplication and Context Value

The advanced, marker-based review mechanism addressed the common CI problem of excessive notification bloat, where multiple, successive micro-pushing operations resulted in lightweight comment updates, rather than repeatedly sending long comment threads to Pull Request pages. The deduplicated update mechanism ensured context integrity was maintained during the inherently asynchronous operations of the commit process, allowing the code review process to proceed more efficiently. Additionally, the AST-based summarization correctly identified cross-reference anomalies and dead-ends early in the codebase's history, well before the integration test process began.

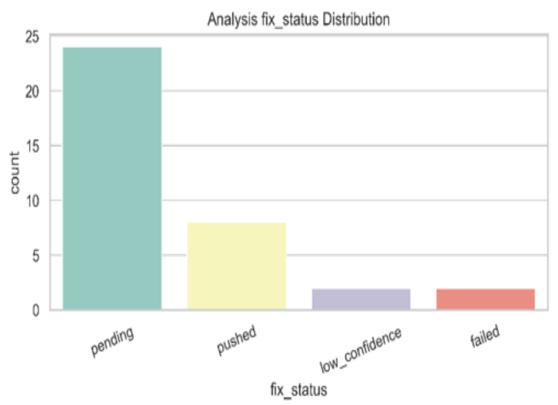

Fig 5: Analysis Fix Status Distribution

Figure 5 illustrates the distribution of the remediation results with respect to runtime error analysis. Each bar represents the number of analysis reports with varying fix statuses, such as fixes successfully pushed, failed attempts, fixes still pending, or results only providing advisory guidance.

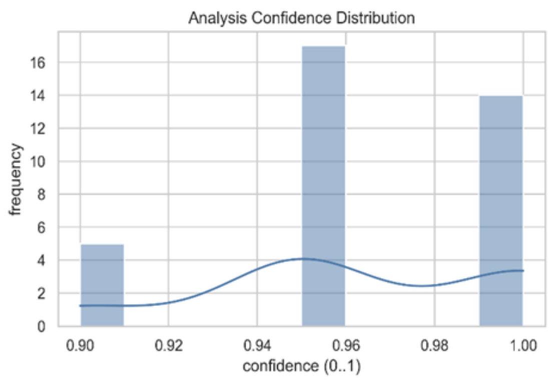

Fig 6: Analysis Confidence Distribution

As shown in Figure 6, the analysis engine assigns a confidence score to each reported runtime error. The histogram indicates how frequently the system produces low, medium, and high confidence diagnoses using replication and weighted scoring. This visualization is helpful in determining the reliability of automated diagnostics and how effectively the confidence gating mechanism is used.

VI. DISCUSSION

The primary architectural advancement presented in AutoCure is not solely isolated to a rudimentary evolution of generative AI deduction capabilities, but specifically relies on the methodological, reliable application of deterministic parsing artifacts acting directly upon systemic automated operations. Structural syntax tree representations establish an incredibly stable, intermediate logical layer fundamentally bridging highly erratic runtime environment telemetry directly with secure, deterministic version control outcomes.

By heavily anchoring generative AI executions firmly within strongly established programmatic verification boundaries specifically parameter validation scaling and AST delta checking computations—the traditional, heavily manual cycle of diagnostic triage is measurably reduced. Software engineering teams theoretically benefit substantially from autonomous incident responses scaling intelligently according to error impact, seamlessly translating initial debugging friction into highly structured, inherently auditable project proposals systematically prepared for required human finalization.

VII. THREATS TO VALIDITY

Some actual-world constraints limit how far our wide-ranging generalizations may be taken, as it is not possible to validate everything exhaustively.

A. Internal Validity

Real fault replication requires that project endpoints be loosely coupled. In cases where you have very tightly integrated project endpoints that must react to remote states of their environment or complex authentication schemes, it is very difficult to replicate faults in a very exact and localized manner. In addition, log parsing is most successful when performed against standard exception formats that are most commonly used in standard web frameworks. In cases of very non-standard or heavily obfuscated stack traces, context extraction is less successful.

B. Construct Validity

But even that is a calculated measure and not a formal proof of optimal patch correctness. Review scores are ultimately a series of matrices generated against a model and must be interpreted against context by a human expert.

C. External Validity

Finally, while our heuristics function very well against standard cases of backend monorepos, very complex paths that traverse a micro-front-end and polyglot back-end networks may not have a completely uninterrupted end to end structural connectivity. In addition, our testing data is strictly contained to a lab environment, and to truly test scaling, a deployment to a thousand-plus developer environment would be required to truly test unhindered growth.

VIII. CONCLUSION

To provide safe, executable debug support within deeply embedded production scenarios, it is necessary to integrate smart, generative reasoning with solid, deterministic verification. AutoCure offers a robust, innovative platform that combines volatile error signals, solid AST understanding, iterative statistical verification, and securely deterministic branch deployments from version control into a cohesive whole.

With solid, mathematically certain remediation parameters paired with thorough, iterative code structural reviews, the system design offers a clear pathway to safe developer productivity while keeping rogue production deployments at bay. It is a step toward a necessary, methodical path to safe, autonomous, and fully integrated self-healing tool sets.

APPENDIX

Project Link: LINK

Recording Link: LINK

ACKNOWLEDGEMENT

The authors would like to extend their heartfelt thanks to everyone and every organization that has contributed in any manner towards the development and completion of this research. The authors would like to extend their special thanks to Pimpri Chinchwad College of Engineering, Pune, for providing the necessary resources and support for the development and implementation of this research. The guidance and expertise provided by the faculty and mentors were essential in the development and refinement of the methodology and analysis of this research. The authors would also like to extend their sincere thanks to the peers and researchers for their constructive discussions and suggestions that helped in the refinement of the methodology and the quality of this research paper as a whole.

The authors would like to extend they're thanks to the developers and researchers for their contributions and development of the publicly available sources that have helped in the development and refinement of the project. The authors would also like to extend their thanks for the support and resources that have helped in the development and refinement of this research paper. This would have been impossible without the contributions and support of the various entities and researchers that have helped in the development and refinement of this research paper. As in any research paper, any mistakes and omissions in this paper are the sole responsibility of the authors.

REFERENCES

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in Proc. International Conference on Software Engineering (ICSE), 2012, pp. 3-13.

[2] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in Proc. International Symposium on Software Testing and Analysis (ISSTA), 2014, pp. 437-440.

[3] Y. Kim et al., "On the opportunities and challenges in engineering self-healing software systems," Information and Software Technology, vol. 166, 2024.

[4] J. Basulto et al., "RepairAgent: An Autonomous, LLM-Based Agent for C/C++ Program Repair," arXiv preprint arXiv:2403.17134, 2024.

[5] Q. Zhang, C. Fang, Y. Xie, and Y. Ma, "A Systematic Literature Review on Large Language Models for Automated Program Repair," ACM Transactions on Software Engineering and Methodology (TOSEM), 2026.

[6] L. Cordeiro et al., "A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification," arXiv preprint, 2025.

[7] O. Ackerman, "Sustainable Code Intelligence: Evaluating AST-Based Batching for Large

Language Model Code Analysis," Bachelor's Thesis, Computing Science, Radboud University, 2025.

[8] M. Bruntink et al., "Automating Code Review: A Systematic Literature Review," ResearchGate preprint, 2025.

[9] T. Smith et al., "Revisiting Code Similarity Evaluation with Abstract Syntax Trees," in Proceedings of the Association for Computational Linguistics (ACL), 2024.

[10] Tree-sitter Developers, "Tree-sitter: an incremental parsing system for programming tools," 2024. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[11] S. Ramírez, "FastAPI Architecture, Background Tasks, and WebSockets," Tiangolo, 2024. [Online]. Available: https://fastapi.tiangolo.com

[12] N. Vidács et al., "Compartmentalization-Aware Automated Program Repair," arXiv preprint arXiv:2603.09544v1, 2024.

[13] Pydantic Developers, "Pydantic validation and BaseModel for robust Python applications," 2024. [Online]. Available: https://docs.pydantic.dev

[14] K. Jin et al., "Towards Secure Code Generation with LLMs: A Study on Common Vulnerabilities," IEEE Transactions on Software Engineering, 2025.

[15] AutoCure Development Team, "AutoCure core architecture modules and AST heuristics: src/main.py, src/services/ast_service.py," 2026.