Journal of Computer Science and Technology Studies
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

JCSTS
AL-KINDI CENTER FOR RESEARCH AND DEVELOPMENT

| RESEARCH ARTICLE

Context-Enriched Logging for Intelligent Code Fix Suggestions Using Large Language Models

Maheswara Kurapati
Independent Researcher, USA
Corresponding Author: Maheswara Kurapati, **E-mail**: mahesh.kurapati.1906@gmail.com

| ABSTRACT

Software systems increasingly depend on runtime logs for problem diagnosis, yet traditional logging lacks depth for automated analysis. Introducing a breakthrough framework that transforms source code to generate context-rich logs containing precise metadata - file locations, function signatures, line positions, and failure details. This metadata creates direct linkages between operational failures and their code origins. Built on language-neutral architecture using intermediate representation techniques, the system works across Java, Python, C#, and other languages within heterogeneous environments. At its core, the framework employs specialized machine learning that analyzes integrated failure contexts to pinpoint root causes and suggest tailored code fixes. Testing activities conducted across web services, embedded systems, including IoT and RIoT devices, mobile applications, and distributed platforms exhibited significant improvements for complicated defects that formerly required considerable manual analysis. And while these solutions primarily address the immediate challenge of diagnosis, they also provide a foundation to develop next-generation systems that perform self-repair while requiring minimal human intervention.

| KEYWORDS

Context-Rich Logging, Code Refactoring Automation, Language Model Diagnostics, Programmatic Fix Synthesis, Software System Observability

| ARTICLE INFORMATION

ACCEPTED: 01 October 2025 **PUBLISHED:** 26 October 2025 **DOI:** 10.32996/jcsts.2025.7.11.3

1. Introduction

Modern software reliability hinges increasingly on observability - a concept transcending basic monitoring by delving into runtime behavior through systematic data capture. While logs continue to serve a critical forensic role for engineers today when debugging complicated system failures, logs have not kept pace with the fundamental role they play, and as such, remain surprisingly rudimentary.

For decades, developers have inserted print statements by hand to focus their attention on execution flows. Eventually, logging frameworks were developed to try to bring a little order to the chaos, yet they remain stuck in a chasm between capturing merely that an error happened and capturing with precision how it happened. Actual examination of production systems reveals troubling inconsistencies - developers typically add logging statements reactively, focusing narrowly on immediate debugging needs rather than creating comprehensive execution records for future analysis.

The typical open-source project contains a patchwork of logging approaches varying not only between modules but even within single files. Such irregularity creates substantial barriers when attempting to correlate runtime behaviors with underlying code. Messages frequently lack essential context about their origination point, execution environment, and the specific parameter values that triggered exceptional conditions. D. Yuan's landmark examination of open-source logging practices (2012)

documented these patterns extensively, revealing how inconsistent severity classifications and contextual detail severely limit diagnostic capabilities.

This contextual void becomes even more troublesome as software architectures become more distributed and heterogeneous. When a failure appears in production, technical personnel must painstakingly rebuild execution flows across system boundaries where no clear breadcrumbs exist connecting the presenting symptoms back to an eventual root cause. The diagnostic process becomes particularly fraught when systems incorporate multiple programming languages or execution frameworks, each implementing distinct error reporting mechanisms without standardized metadata conventions. He's investigation into automated log parsing (2018) quantified these challenges, demonstrating how inconsistent formatting hampers machine-based analysis.

The proposed framework tackles these fundamental limitations through automated source code transformation. By systematically refactoring existing codebases, it injects enhanced logging that captures rich contextual metadata - file identifiers, precise line positions, function signatures, standardized error taxonomies, and comprehensive state information. This metadata creates unambiguous connections between runtime events and specific code locations, establishing the foundation for intelligent analysis.

The framework delivers four significant advancements: first, a language-neutral approach to automatic code enhancement; second, seamless integration between static structure and runtime behavior; third, novel application of language models for diagnostic reasoning; and fourth, rigorous evaluation methodology spanning diverse software categories. By addressing the contextual poverty of conventional logging, this work establishes a pathway toward systems capable of self-diagnosis and autonomous repair.

Subsequent sections detail the framework architecture, code transformation techniques, machine learning analysis pipeline, and performance evaluation across representative software domains and failure scenarios.

2. Framework Architecture

The architecture of the context-aware logging framework is designed around the concepts of modularity and extensibility, allowing it to integrate without significant adjustments to existing software development lifecycles. The system is layered to separate instrumentation logic from the analysis portions of the system, and each can evolve independently of the other as requirements and technologies change. At a high level, the framework has 4 main components: a code analyzer, a log enrichment engine, a runtime context collector, and an LLM inference service that relies on a large language model. The components operatively combine to create and sustain the conversion of traditional logging concepts into a context-rich observability approach that connects runtime behavior and source code knowledge. Research on event logs for software failure analysis has demonstrated that structured approaches to log enhancement can significantly improve failure detection rates and reduce diagnostic time, particularly when the enrichment methodology is systematically applied across distributed system components. Studies examining rule-based approaches for failure analysis have shown that even modest improvements in log context quality can yield substantial gains in automated diagnostic accuracy, highlighting the importance of architectural decisions that prioritize context preservation throughout the observability pipeline [3].

The log enrichment methodology represents the central innovation of the framework, implementing a systematic approach to augment traditional log statements with rich contextual metadata while preserving backward compatibility with existing log processing infrastructure. The framework does not try to remove any existing logging systems but will capture logging calls at compile-time and runtime to provide additional context. The model utilizes aspect-oriented programming principles to separate the cross-cutting concerns of logging, which are fundamental to the function of software, and the main business logic, which is thematic in nature. This distinction allows the entire codebase to improve log quality without altering primary log functions. The enrichment pattern uses template-based concepts to promote a standard structure in enriched log messages while still presenting language-specific log practices and transcending those differences. Empirical studies on software failures have revealed that failure diagnosis is often hindered not by a lack of logging but by insufficient contextual information within those logs. Analysis of production system failures has shown that event logs with richer contextual metadata enable more effective rule-based diagnosis approaches, allowing for automated correlation between symptoms and underlying causes even in complex distributed systems where failure propagation paths are not immediately obvious [3].

Approach	**Key Benefits**	**Limitations**
Traditional Logging	Minimal implementation overhead, familiar developer experience	Lacks contextual metadata, difficult for automated analysis

Structured Logging	Standardized formats, improved machine parseability	Limited connection to source code context, static field definitions
Context-Enriched Logging	Bidirectional code traceability, automated enrichment	Higher storage requirements, initial setup complexity

Table 1: Comparison of Logging Enhancement Approaches. [3]

Metadata collection mechanisms form the foundation of the contextual enrichment strategy, capturing critical information about the execution environment and code structure at the point of log generation. The framework implements both static and dynamic collection methods to ensure comprehensive context capture. Static metadata—such as source file names, line numbers, and function identifiers—are extracted during the compilation phase through abstract syntax tree (AST) analysis and compiler hooks. Dynamic metadata—including callstack information, thread identifiers, and execution timestamps—is gathered at runtime through lightweight instrumentation of logging call sites. To minimize performance overhead, the framework employs selective instrumentation techniques that prioritize collection points based on their diagnostic value, focusing particularly on error handling paths and critical system boundaries. Research examining the evolution of logging statements in long-lived software projects has identified frequent changes to logging implementations that compromise their diagnostic value over time. Studies analyzing the stability of logging statements across software versions have found that contextual metadata is particularly vulnerable to degradation during maintenance activities, underscoring the need for systematic approaches to metadata collection that can withstand evolutionary changes to the underlying codebase [4].

Error code standardization represents a crucial element in making logs machine-parseable and amenable to automated analysis. The framework implements a hierarchical classification scheme for error conditions, mapping domain-specific exceptions and error states to a unified taxonomy. This process of standardization occurs at two levels: during code analysis, in which it identifies errors and patterns in error handling, and at runtime, where it dynamically maps exceptions to a standard error code. The classification framework is extendible, so that development teams can identify errors in the context of their domains while still aligning with the core taxonomy. The hierarchical approach enables both broad pattern recognition across system components and fine-grained diagnosis of specific failure modes. Longitudinal studies of logging practices have revealed inconsistent error representation as a significant barrier to automated analysis, with the same conceptual error conditions often represented differently across system components or after code maintenance. Research on logging statement evolution has demonstrated that standardized error coding schemes significantly improve the stability of error representations across software versions, enabling more reliable automated analysis even as the underlying code evolves [4].

Failure parameter capture and representation complete the context enrichment pipeline, ensuring that the specific values and conditions associated with errors are preserved in the logs for comprehensive analysis. The framework is designed to provide intelligent parameter extraction that recognizes key variables in error handling code paths and automatically adds their values to enriched log messages. The framework uses a serialization mechanism to help manage the diversity of data types and structures while effectively balancing information density and readability to ensure that more complex objects can be represented in a meaningful manner, yet do not become overly verbose. Protection of sensitive data is integrated into the collection of parameters with customizable redaction to preserve private data while retaining diagnostic value. The parameter representation follows a structured format designed for machine parseability, enabling automated analysis systems to extract and correlate values across log entries without requiring complex natural language processing. Studies of logging statement modifications have identified parameter selection as one of the most frequently changed aspects of logging statements, reflecting developers' evolving understanding of which contextual information is most valuable for diagnosis. Research analyzing the relationship between log quality and diagnostic effectiveness has shown that comprehensive parameter capture—when implemented with appropriate serialization strategies—substantially improves both manual debugging efficiency and the accuracy of automated analysis tools [4].

3. Source Code Refactoring for Enhanced Observability

Pinpointing ideal locations for enhanced logging demands sophisticated code inspection techniques beyond simple pattern matching. The framework begins by constructing abstract syntax trees from source files, creating a navigable map of program structure. This initial parsing phase yields a baseline representation subsequently enriched through control flow graphing, which traces execution pathways through the application logic. Particular attention focuses on error handling constructs, component interfaces, and state-modifying operations – areas where additional context delivers maximum diagnostic value.

Call graph construction reveals cross-component relationships often obscured in large codebases, highlighting interaction points prone to communication failures. The analyzer carefully examines existing log statements, evaluating their contextual

completeness against empirical heuristics derived from production system studies. When flagging inadequate logging patterns, the system identifies missing environmental context, absent parameter values, or imprecise error classifications.

Examination of widely used open-source projects revealed alarming inconsistencies in logging implementation. Developers frequently add log statements only after experiencing production failures, resulting in fragmented coverage. The same functional events often appear with wildly divergent formats across different modules. Common problematic patterns include the deceptive "log-and-throw" sequence, producing duplicate records and contextually barren messages lacking essential state information needed for proper diagnosis [5].

The transformation engine forms the operational heart of the refactoring framework, applying structured modifications to the code model while preserving original program semantics. Working from the enriched syntax representation, it implements context-aware modifications that either enhance existing log statements or insert entirely new ones at critical junctures. The enhancement process preserves original message content while wrapping it in structured templates capturing file identifiers, line positions, thread context, and relevant variable states.

When inserting new log statements, the engine carefully adopts project-specific formatting conventions to maintain stylistic consistency while incorporating enriched contextual metadata. Change traceability remains a priority, with the system generating comprehensive before/after comparisons for developer review. Static verification confirms each transformation preserves functional equivalence, preventing inadvertent behavior changes. Field studies demonstrate that exception handling blocks benefit most dramatically from automated enhancement, as these critical regions frequently lack sufficient contextual information despite their diagnostic importance. While human developers typically focus on improving message readability, automated tools excel at incorporating structured metadata elements that humans often overlook [5].

Language neutrality represents a fundamental design principle, achieved through an intermediate representation architecture that decouples core analysis logic from language-specific details. Rather than implementing separate processors for each programming language, the framework normalizes diverse syntaxes into a unified model. This abstraction layer enables consistent analysis across polyglot environments while language-specific adapters handle parsing and code generation for each supported syntax.

Technique	**Implementation Approach**	**Application Context**
AST-based Transformation	Parse code to AST, modify nodes, regenerate code	Language-specific refactoring requires precise grammar
Aspect-oriented Instrumentation	Define cross-cutting concerns, inject at compilation	Framework-level integration with minimal code changes
Template-based Replacement	Pattern matching against code templates, substitute enriched versions	Legacy codebase migration with minimal risk

Table 2: Source Code Refactoring Techniques for Enhanced Observability. [5]

Current implementation supports Java, C#, Python, JavaScript, and Go through dedicated adapter modules implementing standardized interface protocols. Each adapter encapsulates language-specific idioms for logging, exception handling, and code organization patterns. The intermediate representation approach facilitates rapid extension to additional languages by implementing new front-end adapters without modifying core transformation logic. Independent research confirms that unified abstract models effectively bridge semantic variations between programming paradigms, enabling consistent handling of observability concerns across heterogeneous technology landscapes [6].

Beyond language independence, platform neutrality extends the framework's applicability across diverse deployment environments from embedded devices to distributed cloud architectures. A layered abstraction model isolates environment-specific capabilities within a dedicated adaptation tier, presenting consistent interfaces to the core logging framework. This abstraction accommodates variations in timestamp handling, thread identification, process context tracking, and deployment topology recognition across platforms.

For containerized applications, specialized collectors capture container identifiers, orchestration metadata, and infrastructure context. In serverless environments, the framework adapts to ephemeral execution models by emphasizing cold-start detection and execution context preservation. The adaptation layer employs plugin architecture enabling environment-specific customization without core modifications, supporting specialized scenarios from IoT deployments to real-time systems. Multiple studies confirm that abstracting platform-specific details into normalized representations significantly improves analytical

technique transferability across varied deployment contexts, enhancing diagnostic capabilities in heterogeneous computing environments [6].

4. LLM-Based Failure Analysis and Fix Generation

Connecting runtime events to their exact source code origins marks a breakthrough in automated debugging techniques. This linkage creates a two-way bridge between abstract program logic and real-world execution patterns. At its core, the framework establishes connections by extracting embedded metadata tags from enhanced log messages - precise file locations, function signatures, and line positions serve as anchoring points within the codebase. When logs indicate a failure, the system automatically retrieves relevant source fragments, examining not just the immediate error site but expanding outward to include surrounding function logic, invocation chains, and dependent modules identified through static analysis.

The correlation mechanism handles special cases for different language environments. Compiled languages require additional processing to map optimized binaries back to their source origins using debug symbols. When version differences exist between execution environments and code repositories, approximate matching algorithms establish correspondence despite minor evolutionary changes in the codebase. DeepFL techniques have proven that fusing multiple diagnostic signals dramatically outperforms single-method approaches. Their evaluation across thousands of defects demonstrated how spectrum-based techniques excel at certain fault patterns while mutation testing better identifies others. A strategic combination of these complementary approaches significantly narrows the search spaces for complex bugs. Particularly illuminating was their demonstration that execution traces provide critical temporal context missing from static analysis, allowing precise isolation of failure conditions that manifest only through specific interaction sequences [7].

Language models drive the diagnostic reasoning pipeline, applying sophisticated pattern recognition to identify subtle causal relationships within failure data. Rather than implementing a monolithic analyzer, the system employs staged processing that progressively refines understanding. Initial classification modules categorize observed symptoms according to established failure taxonomies, activating specialized analysis pathways optimized for different problem classes. The core analyzer applies carefully engineered prompting techniques that guide language models toward causal inference rather than simple association. These specially formulated prompts incorporate both temporal sequencing and structural relationships, enabling the model to trace effect chains backward to root triggers.

When confronting distributed failures spanning multiple components, decomposition strategies break complex problems into manageable segments before synthesizing comprehensive explanations. Throughout the analysis, the system maintains explicit uncertainty tracking, assigning confidence levels to different hypotheses rather than projecting false certainty. Deep neural approaches to fault localization have demonstrated a remarkable ability to learn subtle execution patterns indicative of specific defect types. Particularly revealing was the discovery that models trained simultaneously on coverage metrics, mutation results, and slice-based analyses develop representational power exceeding any individual technique. Neural architectures with hierarchical attention mechanisms proved especially adept at modeling component relationships in large systems, capturing how failures propagate through dependency chains [7].

Component	Primary Function	Integration Points
Context Integrator	Unifies logs, code, and execution data	Source repositories, log aggregation systems
Causal Analyzer	Identifies relationships between symptoms and root causes	Error taxonomies, historical incident database
Fix Generator	Creates contextually appropriate code solutions	CI/CD pipelines, code review systems

Table 3: LLM Analysis Pipeline Components. [7]

Fix generation builds upon diagnostic insights to produce viable code modifications addressing identified issues. The generation strategy employs a hybrid methodology combining pattern-based templates with generative synthesis. For common error categories, parameterized solution templates encode proven remediation strategies, instantiated with specific values from the current context. Novel problems trigger generative pathways where language models create custom solutions informed by

surrounding code patterns, project-specific conventions, and domain best practices. Fix scope varies intentionally from targeted patches for isolated defects to architectural recommendations addressing systemic weaknesses.

Practical applicability remains paramount throughout generation - suggested changes must maintain backward compatibility and minimize disruption to dependent systems. The generation process considers operational constraints beyond mere correctness, including performance characteristics, security implications, and maintainability factors. For distributed fixes affecting multiple code locations, coordination mechanisms ensure consistent application across component boundaries. CoCoNuT research demonstrated conclusively that ensemble-based neural translation models dramatically outperform individual approaches for program repair tasks. Their comparative analysis showed how models incorporating both syntactic structure and semantic context generate fixes that maintain stylistic and functional coherence with existing codebases. Particularly striking was their finding that diverse model combinations explore solution spaces more thoroughly than single-strategy approaches, identifying viable fixes that individual models consistently miss [8].

Quality assurance for generated fixes involves multi-dimensional evaluation, ensuring both technical correctness and practical applicability. Rather than relying on simplistic metrics, the validation pipeline applies graduated filtering across increasingly stringent criteria. Initial screening employs static analysis techniques to verify syntactic validity and type compatibility, quickly eliminating fundamentally flawed candidates. Surviving solutions undergo dynamic validation through targeted test execution, symbolic evaluation of execution paths, and guided fuzzing to stress-test edge cases.

Beyond functional correctness, evaluation examines broader quality aspects including complexity measurements, pattern consistency with surrounding code, and adherence to project-specific conventions extracted from the codebase. Recognizing inherent uncertainty in automated repair, the system presents multiple viable alternatives with explicit trade-off analysis rather than forcing single recommendations. Each suggestion includes a detailed rationale explaining causal connections between the fix and the underlying issue, enhancing developer trust through transparency. Neural machine translation research has conclusively established that sophisticated validation mechanisms represent an essential counterbalance to the creative power of generative models. Their experiments with diverse model ensembles demonstrated how expanded solution spaces require correspondingly advanced filtering techniques. Particularly influential was their finding that multi-stage validation pipelines achieve an optimal balance between thoroughness and computational efficiency, applying lightweight filters broadly before concentrating intensive validation on promising candidates [8].

5. Evaluation and Results

Testing the context-enriched logging system demanded examination across various dimensions to validate both technical merit and practical value. Three complementary evaluation approaches provided comprehensive insight: laboratory testing with artificial defects, historical analysis of past incidents, and real-world implementation with industry partners. The controlled experiments utilized benchmark applications spanning diverse technology stacks - from web services to embedded devices, mobile platforms to database engines, and distributed processing frameworks. Each target underwent instrumentation with enhanced logging capabilities, followed by systematic fault introduction based on empirically observed failure patterns from production systems.

Historical validation examined past incidents from established open-source projects, applying enhanced analysis to preserved diagnostic data and code versions to determine if faster diagnosis would have resulted. Field testing engaged professional teams across banking, telecommunications, and healthcare sectors who integrated the framework into active codebases and monitored resolution metrics over extended periods. Standardized protocols governed all testing phases, ensuring consistent environmental configuration, controlled inputs, and objective measurement techniques. Instrumentation research from Jin's team demonstrated how critical sampling strategy selection becomes when balancing diagnostic power against performance overhead. Their cooperative concurrency bug isolation work established that statistical approaches detect subtle timing issues while maintaining acceptable resource utilization - particularly crucial for production deployment, where performance impact faces strict limits. Evaluating random, coverage-guided, and adaptive sampling techniques revealed that carefully designed sparse instrumentation detects numerous defects with remarkable reliability while minimizing system impact. These findings directly influenced experimental design decisions, focusing instrumentation at strategic code locations rather than blanket coverage [9].

Measuring framework effectiveness required multidimensional metrics covering the entire observability-diagnosis-repair cycle. Log quality assessment examined contextual completeness by calculating captured diagnostic elements against theoretically optimal information, contextual precision through diagnostic value relative to total log volume, and transformation reliability by verifying semantic preservation in modified code. Diagnostic capability metrics included time-to-diagnosis measurements, accuracy percentages for root cause identification, and search space reduction comparing potential fault locations against traditional techniques. Fix generation quality metrics, tracked compilation success rates, test passing percentages, and developer

acceptance figures for suggested remediations. System impact measurement carefully documented compilation time changes, runtime overhead across various workloads, storage requirements for enhanced logs, and computational costs for analysis operations.

Qualitative assessment incorporated structured interviews with professional developers regarding perceived utility, workflow compatibility, and learning requirements. Benchmark comparisons against existing solutions provided contextual reference points, with particularly notable improvements observed for multi-component defects spanning system boundaries. Jin's experiments with concurrent program analysis established that hybrid approaches combining static inspection with targeted dynamic sampling achieve superior results compared to single-method techniques. Their statistical model evaluation demonstrated how integrating temporal, spatial, and semantic evidence produces substantially more accurate failure prediction than any individual signal alone. These findings validated the multi-dimensional assessment strategy employed throughout evaluation phases [9].

Metric Category	Key Indicators	Measurement Approach
Log Quality	Contextual completeness, information density	Automated metadata extraction analysis
Diagnostic Efficiency	Mean time to diagnosis, search space reduction	Controlled fault injection experiments
Fix Effectiveness	Compilation success rate, test pass rate	Developer acceptance studies

Table 4: Performance Metrics for Observability Framework Evaluation. [9]

Domain-specific case studies revealed fascinating effectiveness patterns across different technology landscapes. Web service deployments showed exceptional results for diagnosing race conditions and resource contention scenarios, leveraging enriched context to correlate seemingly independent events across service boundaries. Embedded implementations delivered modest speed improvements but dramatic log volume reductions - critical for storage-constrained environments with limited transmission bandwidth. Mobile applications benefited primarily from cross-platform consistency features, unifying diagnostics across native and framework-based components that traditionally required separate analysis approaches.

Database deployments showed particularly strong fix acceptance rates, with context-aware suggestions proving especially effective for transactional and concurrency issues. Distributed systems presented both maximum challenges and greatest improvements, with cross-component correlation capabilities diagnosing complex emergent behaviors previously requiring extensive manual investigation. A telecommunications case demonstrated remarkable effectiveness when the framework identified subtle interactions between connection management and load distribution mechanisms, causing intermittent message loss under specific traffic patterns - a problem that had persisted for months under traditional diagnosis techniques. Parnin's research examining actual debugging tool usage delivered crucial insights about practical utility factors. Their studies comparing different diagnostic aids revealed that alignment with developer mental models significantly impacts adoption rates - tools requiring workflow adjustments face steeper adoption barriers regardless of theoretical capabilities. Developer behavior observations documented hypothesis-driven diagnostic approaches, suggesting that tools supporting this natural process achieve better acceptance. Usage pattern analysis demonstrated that immediate perceived benefit predicts continued adoption more reliably than theoretical power [10].

Framework limitations emerged clearly through evaluation, highlighting constraints and future research priorities. Scalability challenges appear in extremely large systems where contextual data volumes can overwhelm both storage infrastructure and analysis pipelines. Future work will explore adaptive detail adjustment techniques that dynamically modulate information capture based on detected anomaly patterns, preserving comprehensive data only for suspected failure paths. Fix generation accuracy remains limited for defects requiring specialized domain knowledge beyond captured code and logs, particularly business logic issues versus technical implementations. Addressing this limitation requires knowledge graph integration, incorporating external documentation, issue histories, and domain models into the analysis process.

Privacy concerns cause other challenges stemming from an increased contextual log that may inadvertently enter into sensitive information without adequate redaction approaches. Ongoing work examines the ability of privacy-reserved techniques such as differential privacy and homomorphic encryption to enable a count-based approach for analyzing log data without compromising protected data. Practical implementation difficulties include that there will be a non-trivial initial setup, potentially complicated in a polyglot environment using multiple technology stacks. Streamlining setup processes through enhanced automation and incremental adoption pathways represents a crucial improvement area. Perhaps most significantly, current language models still struggle with specialized technical domains requiring deep knowledge or specialized reasoning patterns.

This limitation suggests promising research directions in domain-specialized models fine-tuned for specific technology ecosystems and architectural paradigms. Parnin's empirical work identified substantial gaps between theoretical capabilities and practical utility in development workflows. Their studies documented how developers struggle to incorporate tool outputs into diagnostic processes, particularly when reasoning remains opaque. Fault localization technique assessment revealed that even technically superior approaches face adoption challenges when misaligned with mental models or established practices. Investigation into developer needs during debugging demonstrated that comprehensive context about program state and execution flow often provides more value than declarative fault identification [10].

Conclusion

Context-enriched logging represents a fundamental shift in software diagnostic capabilities. Traditional logs primarily served human readers, offering limited utility for automated analysis. By systematically injecting rich contextual metadata and establishing direct links to source code, this framework transforms logs into comprehensive diagnostic foundations. Language neutrality ensures applicability across technology ecosystems from embedded devices to cloud platforms, while platform-independence mechanisms maintain consistent functionality regardless of deployment environment. Field testing confirmed particularly strong results in distributed architectures where traditional methods typically struggle with complex interaction patterns. Notable limitations emerged during large-scale system evaluation, where metadata volume sometimes overwhelms storage infrastructure, suggesting future work in adaptive detail adjustment. Similarly challenging are defects requiring specialized domain knowledge beyond captured code context, indicating potential for knowledge graph integration with external documentation sources. Nevertheless, the framework establishes important infrastructure for increasingly autonomous diagnostic systems. As development methodologies evolve from reactive debugging to more proactive observability, context enrichment is vital for intelligent analysis. In the end, this approach illustrates a future envisioned for self-healing systems where they detect defects and emergency repairs without significant human involvement and either change how engineers develop systems or how they maintain complex software ecosystems.

Funding: This research received no external funding.
Conflicts of Interest: The authors declare no conflict of interest.

References

[1] Ding Yuan et al., "Characterizing logging practices in open-source software," IEEE Xplore, 2012. https://ieeexplore.ieee.org/document/6227202
[2] Shenghui Gu et al., "Logging Practices in Software Engineering: A Systematic Mapping Study," IEEE Xplore, 2022. https://ieeexplore.ieee.org/document/9756253
[3] Marcello Cinque et al., "Event Logs for the Analysis of Software Failures: A Rule-Based Approach," IEEE Xplore, 2013. https://ieeexplore.ieee.org/document/6320555
[4] Suhas Kabinna et al., "Examining the Stability of Logging Statements," IEEE Xplore, 2016. https://ieeexplore.ieee.org/document/7476654
[5] Boyuan Chen; Zhen Ming Jiang, "Characterizing and Detecting Anti-Patterns in the Logging Code," IEEE Xplore, 2017. https://ieeexplore.ieee.org/document/7985651
[6] Wei Xu et al., "Detecting large-scale system problems by mining console logs," ACM Digital Library, 2009. https://dl.acm.org/doi/10.1145/1629575.1629587
[7] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization," ACM Digital Library, 2019. https://dl.acm.org/doi/10.1145/3293882.3330574
[8] Thibaud Lutellier, et al., "CoCoNuT: combining context-aware neural translation models using an ensemble for program repair," ACM Digital Library, 2020. https://dl.acm.org/doi/10.1145/3395363.3397369
[9] Guoliang Jin et al., "Instrumentation and sampling strategies for cooperative concurrency bug isolation," ACM Digital Library, 2010. https://dl.acm.org/doi/10.1145/1869459.1869481
[10] Chris Parnin, Alessandro Orso, "Are automated debugging techniques actually helping programmers?" ACM Digital Library, 2011. https://dl.acm.org/doi/10.1145/2001420.2001445