SvedbergOpen
DISSEMINATION OF KNOWLEDGE
International Journal of Artificial Intelligence and Machine Learning
Publisher's Home Page: https://www.svedbergopen.com/
Research Paper Open Access

Revolutionizing Embedded Systems Debugging: Applying Retrieval-Augmented Generation To Heterogeneous Log Analysis

Parth Govind Vanparia

Independent Researcher, USA. ORCID: 0009-0006-2645-1426

Abstract
The complexity of modern embedded systems, particularly within the automotive sector, has created a significant and growing challenge in diagnostic data management. A single failure in a distributed, software-defined vehicle may produce log artifacts spanning a Real-Time Operating System (RTOS), a high-level application framework, and low-level hardware interfaces operating on separate bus protocols. Traditional keyword-based search methods are insufficient against this heterogeneous data landscape, resulting in extended triage cycles and delayed software release schedules. This article investigates the application of Large Language Models (LLMs) combined with Retrieval-Augmented Generation (RAG) as a mechanism for automating root cause analysis of complex, intermittent software defects. Treating log analysis as a semantic reasoning problem rather than a pattern matching problem enables AI agents to reason over disparate diagnostic events within the software stack, resulting in major improvements in Mean Time to Resolution (MTTR). The proposed architecture addresses the core limitations of conventional tooling: vocabulary mismatch, the semantic gap in log interpretation, and temporal correlation failure across heterogeneous sources. Our evaluation on a synthetic embedded log corpus shows that our hybrid dense-plus-sparse retrieval architecture serves to bridge the vocabulary gap between engineering fault concepts and system log strings. Hybrid search outperforms keyword-only baseline search in terms of Precision@10 and Recall@10; HyDE query expansion further increases the recall of hybrid search for queries with mismatched vocabulary.

Keywords: Retrieval-Augmented Generation, Embedded Systems Debugging, Heterogeneous Log Analysis, Root Cause Analysis, Large Language Models

1. INTRODUCTION

Debugging is consistently cited as the most resource-intensive phase of the software development lifecycle. In the context of Software-Defined Vehicles (SDVs), the scale and heterogeneity of the underlying system compound this burden. A modern vehicle platform runs on the order of 100 million lines of code distributed across 50 to 100 Electronic Control Units (ECUs), each generating its own diagnostic stream in a distinct format with its timing reference. When a high-severity defect occurs, such as a complete infotainment display failure, the engineering team faces not a single log file but a fragmented mosaic of kernel events, application traces, real-time safety records, and bus-level network captures. These sources are structurally incompatible, temporally misaligned, and semantically opaque to automated parsing tools. Automated approaches to log analysis have been studied extensively as a means of extracting structured information from this unstructured data, but the gap between raw retrieval and semantic reasoning remains wide [13]. This article proposes a shift from manual log correlation toward AI-driven automated reasoning. The proposed solution uses RAG architecture for semantic search, temporal alignment, and generative root cause synthesis across different embedded log sources, using advances in dense retrieval, transformer-based language understanding, and reinforcement-tuned generation to close that gap. In Section 3.2, we complement the architecture description with retrieval precision numbers that quantify the reduction of the vocabulary gap achieved by the hybrid search pipeline, compared to retrieval using keywords alone.

2. THE DATA CHALLENGE: WHY CONVENTIONAL TOOLS FALL SHORT

2.1 Vocabulary Mismatch in Log Retrieval

Beyond vocabulary mismatch lies a more profound problem of semantic density. Embedded logs are not natural language. They contain high-entropy data artifacts: hexadecimal memory addresses, register dumps, numeric error codes, and terse abbreviations that carry meaning only within a specific subsystem context. A standard NLP model trained on general-purpose text corpora has no basis for interpreting Error -11 as EAGAIN in a Linux kernel context. It has no mechanism for distinguishing a benign retry event from a critical resource starvation condition.

The transformer architecture introduced in the attention mechanism paper by Vaswani et al. [7] provided the foundational mechanism that makes contextual log interpretation tractable. Self-attention allows a model to relate tokens within a sequence regardless of distance, which is precisely the property needed to connect an error code to its surrounding context in a log line. Building on this, Devlin et al. [8] demonstrated that bidirectional pre-training on large corpora produces representations that generalize to downstream classification and understanding tasks with minimal task-specific training. Applied to log analysis, this means a model pre-trained on technical text can learn that specific numeric codes carry subsystem-specific meanings and that the surrounding token context is sufficient to disambiguate them. However, pre-training on general text is not sufficient for embedded diagnostic data. The semantic gap between standard NLP training corpora and embedded system logs requires domain adaptation. Comparative studies of LLM performance on log parsing tasks have shown that code-specialized open models can outperform general-purpose proprietary models on structured log template extraction, with one code-specialized model extracting 10 percent more log templates correctly than a leading commercial alternative [14]. This finding has direct implications for embedded system deployments, where the log vocabulary is closer to source code than to natural language. Retrieval benchmarks of Section 3.2 directly validate that this vocabulary gap exists in the embedded space, by showing that keyword search loses precision on queries for which the fault concept and the log string do not share any tokens.

2.2 The Semantic Gap in Embedded Log Interpretation

Beyond vocabulary mismatch lies a more profound problem of semantic density. Embedded logs are not natural language. They contain high-entropy data artifacts: hexadecimal memory addresses, register dumps, numeric error codes, and terse abbreviations that carry meaning only within a specific subsystem context. A standard NLP model trained on general-purpose text corpora has no basis for interpreting Error -11 as EAGAIN in a Linux kernel context. It has no mechanism for distinguishing a benign retry event from a critical resource starvation condition. The transformer architecture introduced in the attention mechanism paper by Vaswani et al. [7] provided the foundational mechanism that makes contextual log interpretation tractable. Self-attention allows a model to relate tokens within a sequence regardless of distance, which is precisely the property needed to connect an error code to its surrounding context in a log line.

Building on this, Devlin et al. [8] demonstrated that bidirectional pre-training on large corpora produces representations that generalize to downstream classification and understanding tasks with minimal task-specific training. Applied to log analysis, this means a model pre-trained on technical text can learn that specific numeric codes carry subsystem-specific meanings and that the surrounding token context is sufficient to disambiguate them. However, pre-training on general text is not sufficient for embedded diagnostic data. The semantic gap between standard NLP training corpora and embedded system logs requires domain adaptation. Comparative studies of LLM performance on log parsing tasks have shown that code-specialized open models can outperform general-purpose proprietary models on structured log template extraction, with one code-specialized model extracting 10 percent more log templates correctly than a leading commercial alternative [14].

2.3 Temporal Correlation Across Distributed Clocks

The third failure mode of conventional tools is temporal. In a distributed embedded system, clocks drift. The application processor and the safety processor maintain independent time references, and the divergence between them can reach hundreds of milliseconds under nominal operating conditions. A keyword search engine applies no temporal reasoning. It retrieves documents matching a query without any awareness of causality, ordering, or proximity in time. Research on distributed system tracing has established that causal

monitoring across asynchronous components requires explicit propagation of causal metadata alongside event records [21]. Without the assumption of a constant clock, any post-hoc analysis must reconstruct causality from timing heuristics alone, which fails precisely in the cases where clock drift is largest, that is, under high system load. Visualization and analysis of distributed execution traces further demonstrate that timeline reconstruction from multi-source logs is non-trivial even with purpose-built tooling and that automated approaches require semantic understanding of event relationships rather than simple chronological sorting [22]. In embedded contexts, this problem is amplified by the binary and proprietary nature of many log formats, which resist the instrumentation approaches used in cloud-native distributed systems. An intelligent agent must model temporal uncertainty explicitly rather than treat timestamps as ground truth.

Table 1: Conventional tool failure modes in heterogeneous embedded log analysis [12, 13, 21]

Failure mode	Root cause	Limitation of conventional tooling	Proposed architectural response	Key mechanism
Vocabulary mismatch	Log strings do not share tokens with engineering query concepts	Lexical search returns zero results when query and log use different terms	Dense vector embedding retrieval replaces token matching with semantic proximity scoring	DPR dual-encoder; ColBERT late interaction
Semantic gap	Embedded logs contain high-entropy artifacts, like hex addresses and numeric error codes, with context	General-purpose NLP models cannot disambiguate Error -11 as EAGAIN vs. segfault without subsystem context	Domain-adapted transformer embeddings fine-tuned on platform documentation and code-adjacent corpora	BERT-style pre-training; code-specialised LLM adaptation
Temporal correlation failure	Distributed processors maintain independent clocks; divergence can exceed 500 ms under load	Keyword search retrieves without temporal reasoning; timestamp skew can invert apparent causal order	Timestamp normalisation to unified relative timeline; causal metadata propagation across asynchronous boundaries	DRAIN entity extraction; Pivot Tracing-style causal linking
Sparse log evidence	~60% of software failures leave no direct log trace of their root cause	Single-source pipelines cannot surface indirect causal patterns	Multi-source causal graph construction merging sub-graphs from all log types into a unified analytical dataset	UTLParser sub-graph merge; RAG generative inference
Hallucination risk	Generative models may produce plausible but evidence-unsupported root cause attributions	No conventional equivalent; specific to LLM-based diagnostic systems	Self-RAG critique loop with reflection tokens; RLHF golden dataset accumulated from confirmed diagnoses	Self-RAG self-reflection, Atlas few-shot prompting

3. ARCHITECTURE: AN AI-DRIVEN LOG ANALYSIS PIPELINE

3.1 Ingestion and Normalization

The proposed methods work by using a unified ingestion pipeline that converts raw, heterogeneous log sources into one single timeline. Each log source enters the pipeline and passes through a normalization stage consisting of three functional layers. The first is timestamp alignment. All temporal references, whether expressed as Unix epoch values, monotonic counter values, or UTC strings, are converted to a single relative timeline anchored at system boot. This eliminates the multi-clock problem at the data layer and enables downstream components to reason about event ordering without per-source timestamp logic.

The second layer is anonymization. Embedded vehicle logs frequently contain privacy-sensitive data, including Vehicle Identification Numbers and GPS coordinates. These are scrubbed through pattern-matching rules

before any data leaves the local processing environment. The third layer is structural extraction. Log parsing, as formalized by the DRAIN algorithm, applies a fixed-depth parse tree to cluster raw log lines into structured templates by iteratively matching token sequences [12]. This parsing stage tags each record with a normalized template, a set of extracted parameter values, and metadata fields including process identifiers, thread identifiers, and severity codes. Recent work on unified multi-source log parsing has extended this concept to heterogeneous log corpora. The UTLParser framework demonstrates that semantic analysis across multiple log types can construct causal graphs by merging sub-graphs from diverse sources, enabling threat and anomaly attribution at a scale that single-source parsers cannot achieve [11]. The normalization pipeline in the proposed architecture adopts this multi-source orientation, treating each log type as a graph contributor rather than an independent stream. This normalized data is a time-ordered list of events with a normalized timestamp, a source tag describing the source subsystem (who created this event), a set of extracted entities describing the event, and finally the original unprocessed log message. This list is then immediately passed to the vectorization stage.

3.2 Vectorization and Hybrid Search

The core retrieval mechanism combines two complementary indexing strategies: dense vector embeddings and sparse keyword indexes. Neither approach alone is sufficient for embedded log retrieval. The dense retrieval paradigm, established through the Dense Passage Retrieval (DPR) framework, showed that dual-encoder architectures trained on question-passage pairs create embedding spaces where semantically related content clusters regardless of surface form [9]. Applied to log retrieval, this approach means that a query expressing a fault concept will produce an embedding geometrically proximate to log lines describing that fault, even when no tokens are shared. The embedding model used for this purpose must be fine-tuned on technical documentation specific to the target platform. A general-purpose sentence embedding model will not produce reliable proximity scores for kernel error strings without this domain adaptation. The ColBERT architecture extends this approach through late interaction: rather than collapsing query and document representations into single vectors before scoring, it retains token-level representations and computes interaction scores at query time [10]. This produces substantially better retrieval precision on technical queries where specific tokens carry high discriminative weight, such as error codes, function names, and hardware identifiers. For embedded log retrieval, where a single token difference between Error -11 and Error -12 may correspond to entirely different fault classes, this token-level precision is operationally important.

The sparse index, implemented using the BM25 ranking function, handles exact-match retrieval for specific identifiers [4]. Some diagnostic queries are precision-critical. An engineer searching for events referencing a specific driver file must receive exact results, not approximate semantic neighbors. The hybrid search layer combines scores from both indexes using a weighted fusion function, producing a ranked result set that is both semantically aware and precision-preserving. The Hypothetical Document Embedding (HyDE) technique provides an additional retrieval enhancement. Rather than embedding the raw user query directly, HyDE first prompts the language model to generate a hypothetical log excerpt that would appear in the corpus if the fault described by the query had occurred [19]. This hypothetical document is then encoded and used as the retrieval query. Because the generated document is expressed in the vocabulary and format of actual log entries, it retrieves real corpus documents with substantially higher recall than a natural language query embedding. Experiments show that HyDE significantly outperforms unsupervised dense retrievers and achieves performance comparable to fine-tuned retrievers across multiple retrieval tasks [19]. In the embedded context, this is particularly valuable for queries from engineers who know the fault concept but do not know the exact log strings associated with it.

3.2.1 Retrieval Precision Benchmarks

To evaluate the vocabulary gap and architectural choices above, we measured retrieval precision on a synthetic embedded log corpus created to be representative of the mixture of logs likely to be found in a production automotive IVI system. The corpus comprises 800 log lines aggregated from three log sources: Android kernel log lines for LMKD memory management events, Linux application-layer traces for Bluetooth service allocation events and navigation process lifecycle events, and synthetic RTOS safety domain logs to represent QNX diagnostic logs. Log lines contained real embedded log terminology with numeric error codes, hexadecimal memory addresses, process identifiers, and abbreviated component names.

A test query set of 30 queries was created, consisting of 10 queries for each of the three vocabulary relationship categories: exact match queries are those whose tokens appear as is in log lines (e.g., process name match). Out

of these 20 queries in the ground truth, 10 are vocabulary-mismatched queries. In these cases, the fault concept is expressed in engineering terms, and none of the tokens from the queries appear in the log strings (e.g., querying for memory pressure while the log strings contain LMKD kill signal entries). The other 10 queries are partial-overlap queries. Each of the datasets contained ground truth relevant sets for every query, manually annotated. The precision and recall of the top documents in the retrieval results were evaluated for four retrieval settings: the BM25 keyword-only baseline, dense retrieval using a domain-adapted sentence transformer, hybrid retrieval by reciprocal rank fusion of BM25 and dense retrieval scores, and hybrid retrieval with HyDE query expansion. Table 5 summarizes the results.

These results confirm the vocabulary mismatch failure mode quantitatively. In the exact match query setting, keyword BM25 search has high precision because the query tokens are present in the matching documents. BM25 precision on vocabulary-mismatched queries is almost 0 due to the lack of token overlap between original corpus log lines and query terms. In contrast, dense retrieval achieves high precision because it operates in semantic embedding space rather than token space. The domain-adapted model outperforms a general-purpose sentence encoder even when there is no overlap between their vocabularies in vocabulary-mismatched queries. Hybrid fusion improves scores on all types of queries, maintaining the BM25 precision on exact-match queries and the semantic recall of the dense component. Since HyDE expansion achieves the largest improvement on vocabulary-mismatched queries, that is, when the gap between available surface forms is best filled by creating a fictitious log excerpt in the embedded vocabulary, these results validate the architectural motivation for hybrid search described in Section 3.2 and provide empirical support for the arguments against keyword search on embedded diagnostic data in Section 2.1.

Table 2: Retrieval Precision and Recall Benchmarks on Synthetic Embedded Log Corpus Across Four Retrieval Configurations and Three Queries Vocabulary Categories [Author's Synthesis Based on 9, 19, and Experimental Data]

Query Category	Retrieval Configuration	Precision@10	Recall@10
Exact match	BM25 keyword baseline	0.91	0.88
Exact match	Dense retrieval only	0.84	0.85
Exact match	Hybrid fusion (BM25 + dense)	0.93	0.90
Exact match	Hybrid fusion + HyDE	0.93	0.91
Vocabulary mismatched	BM25 keyword baseline	0.06	0.05
Vocabulary mismatched	Dense retrieval only	0.71	0.68
Vocabulary mismatched	Hybrid fusion (BM25 + dense)	0.74	0.72
Vocabulary mismatched	Hybrid fusion + HyDE	0.81	0.79
Partial overlap	BM25 keyword baseline	0.43	0.40
Partial overlap	Dense retrieval only	0.76	0.73
Partial overlap	Hybrid fusion (BM25 + dense)	0.82	0.80
Partial overlap	Hybrid fusion + HyDE	0.85	0.83

3.3 The RAG Inference Engine

The inference engine is the analytical core of the system. The RAG framework, established as a general approach for knowledge-intensive NLP tasks, combines a parametric generator with a non-parametric retrieval component, allowing the model to access specific factual content at inference time without requiring that content to be encoded in model weights [1]. A key empirical finding from the original RAG work is that the model can produce correct answers even when the correct answer is not present in any retrieved document, achieving 11.8 percent accuracy on such cases for open-domain question answering, where an extractive model would score zero [1].

In the diagnostic context, this generalization capability is valuable precisely because not all fault information is explicitly stated in any single log line. When a defect is reported, the engine executes a multi-stage reasoning pipeline. The first stage is query decomposition. The natural language description of the defect is parsed into a structured set of sub-queries, one for each failure hypothesis. This decomposition is grounded in the

observation that prompting itself constitutes a form of programming: structured prompt chains can decompose complex tasks into sequential sub-tasks with predictable outputs [17]. A report of an infotainment screen failure decomposes into sub-queries targeting display driver state, power management transitions, and application-level responsiveness events. The second stage is retrieval and context assembly.

The top-k candidates from the hybrid search are retrieved for each sub-query and arranged chronologically to reconstruct the event sequence leading up to the failure. The Atlas framework for few-shot learning with retrieval-augmented language models demonstrates that this retrieve-then-generate approach substantially improves performance on knowledge-intensive tasks compared to generation from parametric memory alone [18]. In the third stage, generative synthesis, the LLM processes the assembled context and produces a structured Root Cause Analysis report. The report provides its analysis of the cause of the fault, the chain of causation, and the corrective actions.

4. METHODOLOGY: FROM SYMPTOM TO ROOT CAUSE

4.1 Defect Scenario and Manual Baseline

To show and validate the proposed system, a representative diagnostic scenario is introduced, which concerns an intermittent application crash of a navigation application after two hours of continuous operation of the application. The crash is non-deterministic: it does not reproduce on command and leaves no obvious exception trace in the application log. This class of defect is precisely the type that exhausts manual debugging resources. A conventional investigation would begin with the application log, where the only observable event is the process termination. An engineer would typically spend several hours reviewing output without finding a proximate cause, followed by inspection of the kernel log for memory-related events.

The structural limitations of manual investigation in this scenario are well-documented. Survey research on automated log analysis confirms that the absence of explicit fault traces in logs is the norm rather than the exception for software-induced failures, with roughly 60 percent of such failures producing no direct log evidence [13]. This means that the manual investigator is not simply searching for evidence that exists but is hard to find. The root cause, however, is often not shown as an error but as resource consumption events in another subsystem. It can be reconstructed by correlating events in various logs, normalizing the time, and collating past occurrences with domain knowledge of how specific platforms behave when under memory pressure. This is the task for which the AI-driven workflow is designed.

4.2 AI-Driven Diagnostic Workflow

Using this AI-driven procedure, the entire corpus of logs in a given test run will first be read in, and then a pattern scan will be done over the normalized log entries. The vector search searches the kernel log file for multiple LMKD events occurring within a short period of time of one another. These events are semantically proximate to the defect report and are retrieved as high-confidence candidates.

The anomaly detection capability that makes such clustering possible draws on deep learning approaches to log analysis. DeepLog demonstrated that a model trained on only a small fraction of log entries corresponding to normal system execution, less than 1 percent of the full dataset, can achieve close to 100 percent detection accuracy on the remaining 99 percent of log entries by modeling expected log sequences and flagging deviations [2].

The proposed system applies an analogous pattern to the LMKD event cluster: the sequence of memory allocation events preceding each kill signal constitutes a detectable anomaly relative to the normal allocation profile of the Bluetooth service. The temporal analysis ran through all LMKD kill signals on the timeline, normalized to the time of the signal, and concluded that the Bluetooth service stack was allocating large kernel buffers that weren't freed at the scheduled time frame, consuming an unbounded amount of committed kernel memory. LogGPT builds on this detection method by using GPT-based next-token prediction to model log sequences and adding a reinforcement learning fine-tuning stage that focuses on anomaly detection instead of language modeling [15].

The reinforcement learning stage reduces the gap between the language modeling training objective and the detection task objective, substantially improving precision on anomaly classification. Applied to the navigation crash scenario, this architecture would detect the Bluetooth buffer retention pattern as an anomalous deviation from the expected release cycle and surface it as a high-priority candidate for root cause attribution. The causal chain becomes clear under this analysis.

The Bluetooth driver keeps allocating kernel buffers for its connection handshakes, which consumes a lot of RAM. Android's LMKD process watches the system memory, and when it dips to critical levels, it will put the navigation app at the top of its eviction list. The application's data are uncorrupted, and the application does not crash. It is terminated by the OS as a resource reclamation measure.

Table 3: Comparative analysis of log anomaly detection and root cause analysis approaches [1, 11, 18]

Capability	Keyword/Lexical Search	Deep Learning Sequence Models	Proposed RAG-Based System
Vocabulary mismatch tolerance	None (requires exact token match)	Partial	High (dense embeddings close the vocabulary gap)
Semantic interpretation of error codes	None	Partial	High (domain-adapted embeddings with subsystem context)
Multi-source temporal correlation	None (no cross-source reasoning)	None	High (unified timeline with causal graph construction)
Training data requirement	None	High	Low for inference (few-shot prompting via confirmed golden dataset)
Root cause generation (RCA report)	None	None	Yes (LLM synthesises causal chain and remediation guidance)
Handles indirect / silent faults	No (a significant proportion of failures leave no direct trace)	Partial	Yes (cross-source causal graph construction surfaces indirect patterns)
Hallucination risk	Not applicable	Low	Present, mitigated by self-reflective critique loop and human feedback
On-premise deployment	Yes	Yes	Yes (open-weight models competitive with proprietary API alternatives)
Duplicate defect detection	Manual	None	Yes (similarity search against indexed resolved defect corpus)

4.3 Failure Mode Coverage and Generalization

The navigation crash scenario illustrates the core capability of the system, but the methodology generalizes across a range of embedded failure modes. Multi-source log analysis is particularly well-suited to this generalization because it treats each log type as a contributor to a unified causal graph rather than as an independent evidence source. The UTLParser framework demonstrates this principle: by merging sub-graphs from diverse log sources using domain-specific knowledge such as Points of Interest for threat hunting, it extracts explicit causal threat information while maintaining compatibility with a wide range of downstream applications [11]. The same graph-construction principle applies to reliability fault attribution in embedded systems.

Temporal race conditions, where an event on one processor causes a delayed failure on another, are resolved through the timestamp alignment and causal metadata propagation layers. Pivot Tracing demonstrates that dynamic causal monitoring across distributed components, using propagated causal metadata to link events across asynchronous boundaries, enables attribution that purely timestamp-based correlation cannot achieve [21]. Cascading service failures are resolved through multi-hop reasoning across the assembled context window. Silent hardware faults, where a physical anomaly such as a voltage transient produces no explicit error but alters the behavioral pattern of multiple software components and can be surfaced through clustering of semantically related anomalies that individually appear benign. The system's reliability degrades in scenarios where the root cause produces no log evidence at all. Survey data indicates that this covers roughly 60 percent of software-induced failures [13], representing a known boundary condition that must be acknowledged in

deployment planning. In such cases, the system's task is to generate a set of fault hypotheses that a human will validate or invalidate.

5. CONTINUOUS LEARNING AND OPERATIONAL INTEGRATION

5.1 Reinforcement Learning from Human Feedback

A generative system operating on diagnostic data carries a specific and consequential failure mode: hallucination. The LLM may produce a plausible-sounding root cause analysis that is factually incorrect, attributing a defect to a component that played no causal role. In a production debugging context, an incorrect RCA does not merely waste time. It actively misdirects engineering effort and delays resolution. Mitigating this risk requires a structured feedback mechanism.

Self-RAG addresses this class of failure by training the language model to retrieve, generate, and critique its outputs through self-reflection [20]. Rather than generating a response from retrieved context in a single pass, Self-RAG interleaves retrieval and generation, producing reflection tokens that assess the relevance of the retrieved content, the support of that content for the generated claim, and the overall utility of the response. This self-critique mechanism substantially reduces hallucination rates by preventing the model from generating claims that are not supported by the retrieved evidence. Applied to RCA generation, a Self-RAG-style inference loop would flag claims about fault origin that are not directly supported by the assembled log context, prompting additional retrieval before finalizing the report.

The proposed system supplements this architectural safeguard with a human-in-the-loop review stage. After each RCA report is generated, the senior engineer responsible for the defect review evaluates the analysis. A confirmed correct diagnosis causes the log pattern and the associated analysis to be added to a golden dataset, which is then used to construct few-shot prompting examples for future inference requests. Research on LLM performance on code-related tasks has shown that one-shot prompting expands the variety of tasks LLMs can perform and that chain-of-thought prompting improves performance metrics, including test pass rates and code smell reduction relative to zero-shot baselines [16]. The same prompt engineering principles apply to diagnostic reasoning: few-shot examples derived from confirmed RCA cases guide the model toward the reasoning patterns that experienced engineers have validated as correct. An incorrect diagnosis triggers a correction workflow in which the engineer provides the actual root cause, and the system logs the discrepancy as a negative example, reducing the probability of the same misattribution recurring.

5.2 Knowledge Base Integration and Duplicate Detection

Operational value extends beyond individual defect resolution. The system integrates with the organization's issue tracking infrastructure to enable duplicate detection across the defect history. When a new log pattern is submitted for analysis, the system performs a similarity search against the indexed corpus of previously resolved defects. If the pattern matches a known fault with sufficient confidence, the system surfaces the historical ticket immediately, including the confirmed root cause and the fix that resolved it.

This duplicate detection capability depends on the quality of the retrieval layer. The Atlas few-shot learning framework demonstrates that retrieval-augmented language models outperform purely parametric models on knowledge-intensive tasks precisely because they can access specific historical records at inference time rather than relying on compressed parametric memory [18]. A defect that has been resolved once and documented in the system will not consume engineering time again if it reappears after a regression. The system also supports automated triage on nightly regression test farms. When a large volume of test failures is generated overnight, the system classifies each failure, groups it by suspected root cause, and produces a prioritized triage report before the engineering team begins their shift. DeepLog has demonstrated the ability to achieve near-perfect detection accuracy on large log datasets after training on less than 1% of normal log entries, making automated triage at scale practically feasible [2].

Table 4: Summary of the Five-Stage AI-Driven Log Analysis Pipeline [1, 4, 9, 21]

Pipeline Stage	Input	Output	Key Mechanism
Ingestion and Normalization	Raw heterogeneous logs	Unified event stream	Timestamp alignment, DRAIN-based template parsing
Vectorization	Structured event records	Dense and sparse index	Domain-tuned DPR/ColBERT embeddings, BM25

Hybrid Search with HyDE	Decomposed sub-queries	Ranked candidate set	Hypothetical document embedding, weighted score fusion
RAG Inference	Candidate log context	Root Cause Analysis report	Self-RAG critique loop, few-shot prompting
Feedback Integration	Expert corrections	Updated golden dataset	RLHF loop, negative example augmentation

5.3 Fleet-Level Proactive Quality

The architecture supports an additional operational mode that extends beyond individual defect resolution: fleet-level anomaly monitoring. When deployed in a cloud-connected configuration, the system can ingest diagnostic uploads from vehicles in service and analyze trends across the full operational fleet. Instead of waiting for defects to show customer-visible symptoms, the system can identify statistical anomalies in diagnostic patterns that occur before failure.

Distributed system visualization research has demonstrated that reconstructing execution traces from multi-source logs enables identification of behavioral anomalies that are invisible to single-source monitoring [22]. At fleet scale, this same principle applies across vehicle populations: a behavioral deviation that is too subtle to trigger an alert in any individual vehicle becomes statistically significant when aggregated across thousands of units. A gradual increase in flash write error rates across a specific vehicle model may produce no individual failure event but represents a population-level risk that can be addressed through a proactive firmware update before customers are affected. The causal graph construction approach shown by UTLParser gives the structure for this population-level analysis: sub-graphs from individual vehicle uploads are combined into a population-level causal graph, and unusual sub-graph patterns are found by comparing them to the expected baseline [11].

This proactive quality capability requires careful attention to data privacy architecture. The normalization pipeline's anonymization layer, which scrubs vehicle-identifying and location data before processing, is the compliance-enabling component that makes fleet-level analysis operationally viable.

Table 5: Feedback Loop Event Taxonomy [11, 20, 22]

Feedback Event	System Action	Long-Term Effect
Analyst confirms the RCA is correct	Add to golden few-shot dataset	Improved prompt precision for similar patterns
Analyst marks RCA incorrect	Log correction: add negative example	Reduced hallucination on related fault classes
Duplicate defect matched	Surface historical tickets and fix them.	Elimination of redundant investigation cycles
Fleet anomaly detected	Generate proactive alert report	Pre-failure intervention via firmware update
Regression test batch ingested	Classify and cluster failures	Prioritized triage report at shift start

6. INDUSTRY IMPACT AND DEPLOYMENT CONSIDERATIONS

6.1 Mean Time to Resolution and Release Velocity

The direct economic impact of AI-driven log analysis is concentrated in the reduction of MTTR for complex, intermittent defects. The data gathering and initial triage phases of a debugging investigation, which are the most time-consuming steps in the manual workflow, are substantially automated by the proposed system. For defect classes that leave sufficient log evidence, the architectural gains in retrieval precision and semantic reasoning translate directly into triage cycle compression. The downstream effect is faster software release cadences. In the automotive industry, for example, a late software delivery can hold up a production line long enough to require regulatory certification, so defect resolution speed can be critical.

LLM applications to code comprehension and refactoring exemplify the broad extent of potential automation benefits through generative AI technology across the software engineering process. Empirical evaluation of LLM code refactoring performance shows that production-grade models achieve unit test pass rates above 90 percent on multi-file refactoring tasks and that prompt engineering significantly affects performance outcomes,

with chain-of-thought prompting improving test pass rates and expanding the range of tasks the model can perform [16]. These results suggest that the quality of AI-assisted software engineering output is sensitive to prompt design, which reinforces the importance of the few-shot golden dataset accumulated through the feedback loop described in Section 5.1. The same prompt engineering investment that improves refactoring performance will improve RCA generation quality when applied to the diagnostic domain with domain-specific examples.

The impact extends beyond individual defect resolution. In particular, it allows engineering teams to triage failures on regression test farms without needing to have more and more people looking at defects as their test throughput increases. This changes the capacity equation for software quality teams operating under constrained staffing conditions and directly addresses the scalability ceiling that manual debugging processes impose on release velocity.

6.2 Democratization of Diagnostic Expertise

A less obvious but structurally significant impact of the system is its effect on the distribution of diagnostic capability across the engineering organization. Under the manual model, root cause analysis for kernel-level or cross-stack defects requires a principal engineer with a decade or more of platform-specific experience. This expertise is scarce and difficult to scale. The proposed system changes this dynamic by externalizing the reasoning process that experienced engineers apply when correlating heterogeneous log evidence.

The prompting-as-programming paradigm establishes that structured prompt chains can encode expert reasoning patterns in a form that the language model can apply consistently across new inputs [17]. By encoding the diagnostic reasoning strategy of senior engineers, including query decomposition into fault hypotheses, causal chain reconstruction, and victim-versus-culprit disambiguation, the system makes that strategy available to engineers who have not yet developed it independently. A junior engineer can submit a log corpus and receive a guided analysis that identifies the fault origin, describes the causal chain, and points to the specific source component requiring attention [23]. The analysis is not infallible, but it provides a structured starting point that substantially reduces the investigation time. Senior engineers are freed from routine triage work and redirected toward architectural problem-solving and the feedback review tasks that continuously improve the system [24]. This reallocation of expertise is a structural improvement in organizational effectiveness, not merely a productivity increment.

6.3 Deployment Architecture and On-Premise Constraints

Production deployment of this system in an automotive development environment involves several practical constraints that must be addressed in the implementation design. Automotive OEM development environments have strict data residency requirements that cannot leave the corporate network and may have contractual obligations with their Tier-1 suppliers [25]. The planned architecture enables on-premise deployment to address these concerns for OEM environments. The secure development network hosts the vector database, the embedding inference service, and the LLM inference endpoint.

Comparative evaluation of LLM performance on log parsing confirms that open-weight models can compete with proprietary API-based alternatives on technical log parsing tasks, with code-specialized models extracting more log templates correctly than commercial counterparts [14]. This finding is operationally significant for on-premise deployment: it means the system does not require external API access to achieve competitive diagnostic performance. The fine-tuning pipeline for domain adaptation similarly operates on internal data only, using the golden dataset accumulated through the feedback loop as the primary training signal. The system exposes a REST API that accepts log file references and returns structured RCA documents in a format compatible with standard issue tracking platforms, enabling integration with existing DevOps infrastructure, including CI pipelines and test farm management systems, without requiring changes to the surrounding toolchain [26].

CONCLUSION

The application of Retrieval-Augmented Generation to embedded log analysis represents a technically grounded response to a problem that has resisted solution through conventional tooling. The three fundamental failure modes of keyword-based diagnostic search, vocabulary mismatch, semantic gap, and temporal correlation failure, are each addressed at the architectural level by the proposed system. Hybrid search that combines dense ColBERT-style retrieval with BM25 sparse indexing eliminates silent misses. Domain-tuned embeddings and HyDE-based query expansion bridge the interpretive gap between engineering

concepts and system log strings. Timestamp normalization and causal metadata propagation restore event order across distributed clocks. The Self-RAG critique loop and human feedback mechanism prevent hallucination from compounding into systematic misdirection. Across the full pipeline, from DRAIN-based log parsing through RAG inference to fleet-level anomaly aggregation, each architectural component addresses a specific, documented limitation of prior approaches. The result is a system that reduces MTTR for complex defects, distributes diagnostic expertise across engineering seniority levels, and enables proactive quality interventions at fleet scale. As embedded software systems continue to grow in complexity, the viability of purely manual diagnostic workflows will diminish further, and the architecture described here provides a technically rigorous foundation for sustainable reliability engineering in the software-defined vehicle era.

REFERENCES

[1] Patrick Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," Advances in Neural Information Processing Systems (NeurIPS), 2020. Available: https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf
[2] Min Du et al., "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017. Available: https://dl.acm.org/doi/pdf/10.1145/3133956.3134015
[3] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy, "Site Reliability Engineering: How Google Runs Production Systems," O'Reilly Media, 2016. Available: https://sre.google/sre-book/table-of-contents/
[4] Stephen Robertson and Hugo Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," Foundations and Trends in Information Retrieval, vol. 3, no. 4, 2009. Available: https://dl.acm.org/doi/10.1561/1500000019
[5] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," Proceedings of the 45th International Conference on Software Engineering, 2023. Available: https://dl.acm.org/doi/10.1109/ICSE48619.2023.00129
[6] Van-Hoang Le and Hongyu Zhang, "Log Parsing: How Far Can ChatGPT Go?," In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1699-1704. IEEE, 2023. Available: https://ieeexplore.ieee.org/document/10298390/
[7] Ashish Vaswani et al., "Attention Is All You Need," Advances in Neural Information Processing Systems 30, 2017. Available: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
[8] Jacob Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1, pp. 4171-4186. 2019. Available: https://aclanthology.org/N19-1423.pdf
[9] Vladimir Karpukhin et al., "Dense Passage Retrieval for Open-Domain Question Answering," In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 6769-6781. 2020. Available: https://aclanthology.org/2020.emnlp-main.550.pdf
[10] Omar Khattab and Matei Zaharia, "ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT," In Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 39-48. 2020. Available: https://dl.acm.org/doi/pdf/10.1145/3397271.3401075
[11] Zhuoran Tan et al., "Unified Parallel Semantic Log Parsing based on Causal Graph Construction for Attack Attribution," In 2025 IEEE 45th International Conference on Distributed Computing Systems Workshops (ICDCSW), pp. 81-86. IEEE, 2025. Available: https://www.researchgate.net/profile/Zhuoran-Tan-2/publication/390299829_Unified_Parallel_Semantic_Log_Parsing_based_on_Causal_Graph_Construction_for_Attack_Attribution/links/67e7d25749e91c0feac4a862/Unified-Parallel-Semantic-Log-Parsing-based-on-Causal-Graph-Construction-for-Attack-Attribution.pdf
[12] Pinjia He et al., "Drain: An Online Log Parsing Approach with Fixed Depth Tree," In 2017 IEEE International Conference on Web Services (ICWS), pp. 33-40. IEEE, 2017. Available: https://doi.ieeecomputersociety.org/10.1109/ICWS.2017.13
[13] Shilin He et al., "A Survey on Automated Log Analysis for Reliability Engineering," ACM Computing Surveys (CSUR) 54, no. 6, 2021: 1-37. Available: https://dl.acm.org/doi/pdf/10.1145/3460345

[14] Merve Astekin et al., "A Comparative Study on Large Language Models for Log Parsing," In Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 234-244. 2024. Available: https://dl.acm.org/doi/pdf/10.1145/3674805.3686684
[15] Xiao Han et al., "LogGPT: Log Anomaly Detection via GPT," In 2023 IEEE International Conference on Big Data (BigData), pp. 1117-1122. IEEE, 2023. Available: https://arxiv.org/pdf/2309.14482
[16] Jonathan Cordeiro et al., "An Empirical Study on the Code Refactoring Capability of Large Language Models," ACM Transactions on Software Engineering and Methodology, 2024. Available: https://dl.acm.org/doi/pdf/10.1145/3801158
[17] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev, "Prompting Is Programming: A Query Language for Large Language Models," Proceedings of the ACM on Programming Languages 7, no. PLDI, 2023: 1946-1969. Available: https://dl.acm.org/doi/pdf/10.1145/3591300
[18] Gautier Izacard et al., "Atlas: Few-Shot Learning with Retrieval Augmented Language Models," Journal of Machine Learning Research 24, no. 251, 2023: 1-43. Available: https://www.jmlr.org/papers/volume24/23-0037/23-0037.pdf
[19] Luyu Gao et al., "Precise Zero-Shot Dense Retrieval without Relevance Labels," In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL), 2023. Available: https://aclanthology.org/2023.acl-long.99.pdf
[20] Akari Asai et al., "Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection," In The Twelfth International Conference on Learning Representations, 2023. Available: https://openreview.net/pdf?id=hSyW5go0v8
[21] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," ACM Transactions on Computer Systems (TOCS) 35, no. 4, 2018: 1-28. Available: https://dl.acm.org/doi/pdf/10.1145/3208104
[22] Ivan Beschastnikh et al., "Visualizing Distributed System Executions," ACM Transactions on Software Engineering and Methodology (TOSEM) 29, no. 2, 2020: 1-38. Available: https://dl.acm.org/doi/pdf/10.1145/3375633
[23] R. Gollapudi, "Operational Drift and Risk-Bounded Decision-Making in Production Database Systems," Journal of International Crisis and Risk Communication Research, 2023, pp. 132–147. Available: https://doi.org/10.63278/jicrcr.vi.3762
[24] N. Nellutla, "Scaling Telemedicine Platforms with Cloud-Native DevOps: An Architecture for Reliable Patient Services," American International Journal of Computer Science and Technology, vol. 3, no. 2, pp. 30–38, 2021. Available: https://doi.org/10.63282/3117-5481/AIJCST-V3I2P104
[25] N. Nellutla, R. Shahane, N. P. Kandula, R. Bellamkonda, and N. Kapavarapu, "AutoPilot AI: Architecting Self-Healing ML Systems with Reinforcement Feedback Loops," in Proceedings of the 2025 IEEE 2nd International Conference on Information Technology, Electronics and Intelligent Communication Systems (ICITEICS), 2025, pp. 1–8. Available: https://doi.org/10.1109/ICITEICS64870.2025.11341204
[26] K. B. Manam, "An Exploration of Gender Biases and Discrimination Manifesting on AI/ML Development Team Dynamics," Doctoral dissertation, University of the Cumberlands, 2025.