An AI Agent Framework with Elasticsearch for Scalable Post-Silicon Debug Automation

Minuk Lee[1], Hanna Jang[2], Seonghee Yim[3], Youngsik Kim[4]
([1]minuk123.lee; [2]hn01.jang; [3]sh.yim; [4]ys31.kim@samsung.com)

***Abstract*- Post-silicon debug remains a critical bottleneck, often requiring engineers to manually analyze raw hardware data such as scandumps. This paper introduces a novel conversational AI system that automates the entire debug workflow. At the core, a Large Language Model (LLM) orchestrates a suite of specialized agents, including a Scandump Parsing Agent, an Elasticsearch-based Retrieval-Augmented Generation (RAG) module, and a Jira Integration Agent. Unlike pure vector search solutions, our approach leverages Elasticsearch's hybrid search capabilities, which is critical for debug data that requires both semantic understanding and exact keyword matching. This ensures that knowledge captured once is reused effectively. Early pilot deployments demonstrate a substantial reduction in time-to-debug, improved accessibility of historical expertise, and seamless integration into established workflows.**

I. Introduction

As the complexity of modern System-on-Chip (SoC) designs continues to escalate, post-silicon validation and debug have become one of the most significant and unpredictable bottlenecks in the semiconductor development lifecycle [1]. Engineers are tasked with analyzing vast and heterogeneous data artifacts, including kernel logs, system traces, and register dumps, to pinpoint the root cause of hardware failures.

Among these artifacts, scandumps are particularly critical. A scandump provides a complete, low-level snapshot of the chip's entire register bank at a specific clock cycle, typically captured in a raw binary format. While invaluable for its completeness, this raw data is notoriously difficult to interpret without significant prior expertise. Analysis often relies on a scattered collection of per-block parsing scripts, manual heuristics, and the siloed experience of individual domain experts [2].

This traditional approach suffers from several key challenges:

- Repetitive Analysis Bottleneck**:** The current workflow is highly inefficient. During debugging, an issue is often routed among multiple block/IP owners. Each owner must independently execute a chain of dependent tasks: 1) run a general parsing script to convert the raw binary scandump into text, 2) parse this output to isolate block-specific data, and 3) process the isolated data with a specialized script. This data dependency prevents parallel execution and creates a significant time sink.
- Knowledge Fragmentation**:** Expertise and historical debug findings remain isolated in personal notes or fragmented wikis. This prevents effective knowledge reuse.
- Workflow Discontinuity**:** The debug process is fragmented across disjointed tools for parsing, searching, and documenting, leading to significant manual overhead.

To address these challenges, we propose a conversational AI framework that unifies parsing, historical analysis, and workflow integration under a single interactive interface. This system functions as an end-to-end debug assistant that learns from historical data. The central novelty of our work lies in the Retrieval-Augmented Generation (RAG) [3] module based on Elasticsearch [4], a distributed search and analytics engine. We argue that debug data possesses a dual nature, requiring both semantic understanding (e.g., "Why did the system hang?") and exact keyword matching (e.g., "Find cases where PCIE_LNK_STATUS is 0x01"). We leverage the native hybrid search capabilities of Elasticsearch—which combines traditional text search with vector-based similarity—to address this need, providing a solution that is more accurate, scalable, and operationally simpler than systems built on pure vector databases.

II. Background and Related Work

A. Traditional Scandump Analysis

The conventional method for scandump analysis is an inherently redundant, manual process that scales poorly with SoC complexity. A typical debug cycle begins when a hardware failure is reported and a Jira ticket is created with a raw scandump file attached. The initial assignee follows these steps sequentially, as the output of each step is the prerequisite input for the next:

1) Download and Setup: Download the massive raw binary scandump file from the ticketing system to a local or server-side workspace.

2) Run Scandump Parsing: Execute a general script to convert the raw binary file into a human-readable format. This step decodes the binary bitstream and maps it to the official register hierarchy, producing a structured text file that shows the values for each register and field.
3) Locate Specific Data: Manually filter and extract the relevant registers for the suspected IP blocks or functional units from the large parsed text file. This sub-selection is necessary because the full parsed output contains the status of the entire SoC, most of which is irrelevant to a specific debug task.
4) Run Specialized Parsing: Execute block-specific scripts (e.g., check_cpu_status.py) for deeper analysis.
5) Analyze Final Output: Finally, the engineer analyzes this second-level result file to look for symptoms of the root cause.

A critical bottleneck occurs when the issue is routed between teams. If the initial engineer determines the root cause is likely in a different IP (e.g., handing over the case from the CPU owner to the Bus Interconnect owner), the new assignee must repeat the entire process. They must independently download the same scandump from Jira and re-run steps 2 through 4 to verify their specific block's status. As an issue passes through multiple hands—from GPU to Bus to CPU specialists—this serial repetition of data processing creates a compounding delay and massive duplication of effort across the organization.

B. AI in Hardware Verification

There has been growing interest in applying machine learning (ML) and AI to hardware verification [5]. Much of this work focuses on leveraging ML for pre-silicon log analysis, such as anomaly detection in simulation logs or automatic classification of test failures. These approaches are valuable for passively identifying potential issues from vast datasets. However, these systems often act as black-box classifiers, flagging an issue without providing the interactive context needed for deep root-cause analysis. Our work differs by focusing not on passive anomaly detection, but on building an interactive, conversational system. We aim to empower the engineer during the debug process, allowing them to actively query historical data, ask follow-up questions, and iteratively narrow down the problem space, rather than just reviewing a pre-computed anomaly score.

C. RAG and Hybrid Search

RAG is a framework that enhances the accuracy of Large Language Model (LLM) by retrieving relevant documents from an external knowledge base before generating a response. This approach typically relies on specialized vector databases, such as Milvus [6] and Pinecone [7], which perform semantic similarity searches using high-dimensional embeddings.

However, pure vector search often struggles with the high-precision requirements of hardware debug data. To address this, we employ a Hybrid Search strategy within Elasticsearch. This strategy combines k-Nearest Neighbors (kNN) search [8] for semantic context with the Best Matching 25 (BM25) [9] algorithm for textual precision. BM25 is a sophisticated ranking function used by search engines to estimate the relevance of documents to a given search query based on keyword frequency and saturation. While kNN can find conceptually related issues (e.g., "system hang" vs. "stuck"), BM25 ensures that exact identifiers like register names (ERR_STATUS_REG) or specific error codes (0xDEADBEEF) are prioritized. By unifying these two modalities, our framework achieves the synergistic improvements in accuracy.

D. Agentic Workflow and Orchestration

To manage the interaction between the engineer and specialized tools, the system requires an orchestration layer. We employ LangGraph [10], a library designed for building stateful, multi-agent applications with LLM. While the repetitive, low-level tasks of parsing raw data (as described in Section II-A) are performed by the Scandump Parsing Agent—often in an autonomous, proactive mode—LangGraph serves as the decision-making engine for user-driven, reactive tasks. It interprets natural language prompts to determine the optimal execution path, such as triggering a specific analysis flow or retrieving historical data, ensuring a seamless interface between the engineer and the automated backend.

III. Proposed Framework Architecture

Our system is designed as a multi-agent framework orchestrated by a central LLM, as shown in Figure 1. The LLM interprets the engineer's natural language prompts and dynamically routes tasks to specialized agents.

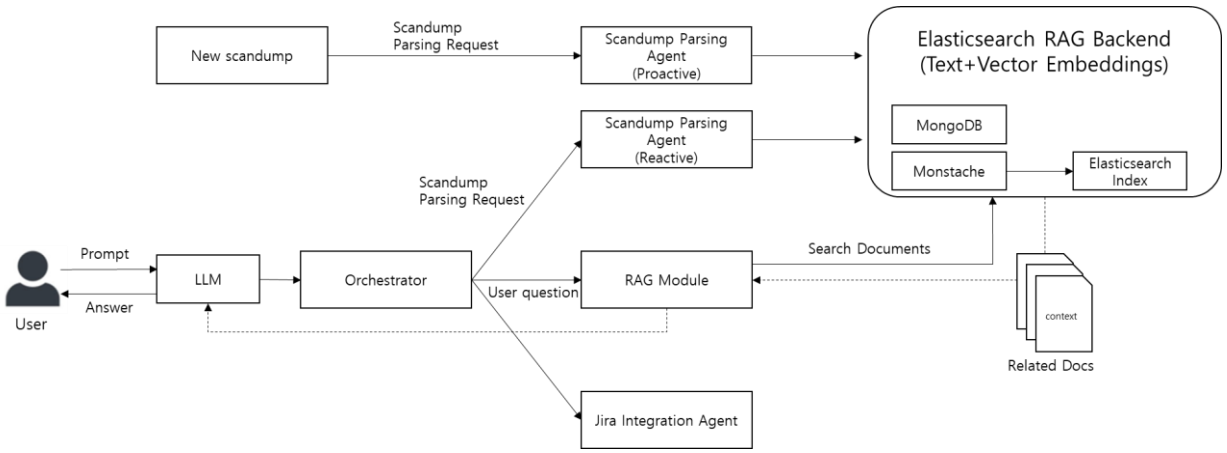

Figure 1 High-level diagram of the proposed AI Agent Framework, showing the LLM Orchestrator coordinating the Scandump Parsing Agent, Elasticsearch-based RAG Module, and Jira Integration Agent.

A. LLM Orchestrator

The LLM Orchestrator serves as the central "brain" of the system. Built using a framework like LangGraph, it maintains the state of the debug session and intelligently invokes the appropriate agent based on the user's intent. For example, a prompt like "Parse this new scandump and find similar issues" would trigger a sequential invocation of the Parsing Agent followed by the RAG Module.

B. Scandump Parsing Agent

The Scandump Parsing Agent is designed to eliminate the redundant cycles described in Section II-A by centralizing the execution and storage of debug artifacts. Rather than developing new parsing logic from scratch, this agent functions as an intelligent wrapper around the pre-existing, domain-specific parsing scripts already established within the organization. It standardizes the execution environment to capture the raw output from these heterogeneous scripts and normalizes it into a unified structured text format. The agent operates in two distinct execution modes to ensure both automation and interactivity:

1) Proactive mode (Automated Pipeline): To break the cycle of repeated downloads and parsing, this mode operates as an automated, event-driven pipeline. Whenever a new scandump is uploaded to a Jira ticket or detected in the repository, the agent automatically orchestrates the execution of all registered IP and block-level parsing scripts. Instead of each engineer running scripts locally, the agent handles these computationally intensive tasks and stores the comprehensive, parsed results in a unified central storage. This creates a single source of truth where the parsed results are mapped to the specific Jira issue.
2) Reactive mode: This mode is triggered by the LLM Orchestrator (LangGraph) in response to a user prompt (e.g., "I just uploaded a local scandump, please parse it now"). The orchestrator identifies the intent and routes the task to the agent to provide immediate, interactive feedback.

Across both modes, the agent consolidates heterogeneous outputs into a unified structured text report. While the Parsing Agent focuses on data transformation rather than autonomous root-cause analysis, it serves as the critical infrastructure that converts opaque binary artifacts into an LLM-readable knowledge base. By doing so, it effectively eliminates the manual data-preparation bottleneck that plagues traditional workflows and provides the essential fuel for the subsequent RAG and orchestration processes. This process transforms transient execution logs into persistent, indexable assets, allowing engineers to actively query and compare current failures against historical patterns stored in the RAG system.

C. Elasticsearch-based RAG Module

This module is the core intellectual property of our framework. It is responsible for indexing the entire collection of pre-analyzed, block-specific reports generated by the Scandump Parsing Agent.

1) The Hybrid Search Imperative for Debug

   Our choice of Elasticsearch is deliberate. Debug data retrieval requires a mix of search modalities to handle both architectural precision and symptomatic ambiguity:

   - Keyword Search (BM25): Essential for finding exact, literal strings. Debugging is often impossible without identifying specific hardware instances or error codes.

     Examples: REG_NAME: "ERR_STATUS", ERROR_CODE: "0xDEADBEEF", BLOCK: "BLK_AUD"
   - Semantic (Vector) Search: Essential for understanding the engineer's intent when different terminology is used for the same failure symptom.

     Examples: "Find issues where a lock-up occurred."

Pure vector databases fail on the first-class keyword requirement. Our module leverages Elasticsearch's ability to natively combine a traditional BM25 inverted index (for keywords) with a dense vector index (for semantics) in a single, unified query.

2) Data and Indexing Pipeline

To enable hybrid search, we designed a robust data pipeline that separates primary storage from the search index, ensuring both data integrity and real-time search capabilities.

- Data Schema: The unified structured text report generated by the Scandump Parsing Agent is first enriched with critical metadata (e.g., project, revision, block_name). This document, containing both the text summary and its corresponding vector embedding, is stored in a primary database (such as MongoDB) which serves as the system-of-record.
- Indexing Strategy: We leverage Elasticsearch as the search index. A synchronization pipeline (such as Monstache) monitors the primary database and automatically indexes new or updated documents into Elasticsearch in near real-time. This decouples the application logic from the search index. The Elasticsearch index mapping is precisely defined to support hybrid search, as shown in Figure 2.

```
{
  "mappings": {
    "properties": {
      "project": {"type": "keyword"},
      "revision": {"type": "keyword"},
      "block_name": {"type": "keyword"},
      "summary_text": {"type": "text"},
      "embedding": {
        "type": "dense_vector",
        "dims": 1024,
        "index": "hnsw"
      }
    }
  }
}
```

Figure 2 Elasticsearch Index Mapping Example

This mapping is critical: metadata fields like project are mapped as keyword for fast, exact-match filtering. The summary_text is mapped as text to leverage the BM25 algorithm. Finally, the embedding is mapped as dense_vector to enable k-Nearest Neighbor (kNN) semantic search.

3) Hybrid Retrieval Strategy

Our retrieval strategy is designed to translate an engineer's natural language query into a precise Elasticsearch query, execute it, and pass the results to the LLM for synthesis.

- Intent Parsing: When a user interacts with the chatbot (LLM Orchestrator), the LLM's primary role is not complex reasoning, but intent parsing. It is prompted to extract key entities from the user's query, such as the core search terms (e.g., "Uncorrectable ECC error") and any filters (e.g., project: "Project1").
- Dynamic Query Construction: The application backend receives these structured outputs from the LLM and dynamically constructs a single, comprehensive Elasticsearch query. This composite query programmatically combines metadata filters, BM25 text search, and kNN vector search. The metadata is applied in a highly-efficient filter context to narrow the search space, while the relevance-based match (BM25) and kNN (vector) queries are combined to calculate a final relevance score. Figure 3 shows a simplified template for this hybrid query.

```
{
  "query": {
    "bool": {
      "filter": [
        {"term": {"project": "Project1"}},
        {"term": {"block_name": "BLK_CPU"}}
      ],
      "should": [
        {
          "match": {
            "summary_text": {
              "query": "Uncorrectable ECC error"
            }
          }
        },
        {
          "knn": {
            "field": "embedding",
            "query_vector": [0.23, ..., -0.11],
            "k": 10,
            "num_candidates": 50
          }
        }
      ],
      "minimum_should_match": 1
    }
  }
}
```

Figure 3 Hybrid Query DSL Template Example

- Retrieval and Synthesis: This hybrid query structure allows Elasticsearch's native scoring to produce a single ranked list of results, blending both keyword and semantic relevance. We do not require a separate re-ranking step; we rely on the unified scoring from the combined query. The top-k retrieved documents (e.g., the summary_text from the top 10 results) are then passed as context to the LLM, which synthesizes them into a coherent, actionable answer for the engineer.

4) Advantages of Elasticsearch as a RAG Backend By using Elasticsearch

We gain several practical advantages over building with a separate vector DB:

- Operational Simplicity: We manage a single, mature platform instead of two separate systems (one for keyword search, one for vector search).
- Proven Scalability: Elasticsearch is already a trusted, industry-standard solution for handling massive volumes of log and text data.
- Maturity and Cost: As a long-standing, open-source platform, it provides robust, cost-effective, and well-supported features for text search, filtering, and aggregation that are essential for a production-grade debug tool.

D. Jira Workflow Integration Agent

To close the debug loop, this agent provides an API interface to the project's issue tracking system (e.g., Jira). After a root cause is identified, the LLM can instruct this agent to "File a ticket for the CPU team with these findings" or "Append these related historical issues (JIRA-123, JIRA-456) to the current ticket."

IV. Implementation and Evaluation

A. Experimental Setup

We evaluated the framework using a dataset derived from a pilot program for a next-generation SoC's post-silicon debugging phase.

- LLM Orchestrator: GPT-OSS-120B
- RAG Backend: Elasticsearch 8.19
- Embedding Model: BAAI/bge-m3
- Dataset: An initial corpus of ~10,000 historical scandump summary files and their corresponding Jira tickets.

B. Illustrative Retrieval Scenarios and Qualitative Feedback

Given the challenge of quantitatively benchmarking complex, interactive debug workflows, we evaluated the system's efficacy through a series of illustrative retrieval scenarios. These scenarios were designed to test the framework's ability to handle the diverse query types encountered by engineers.

1) Scenario 1: Keyword-Dominant Query (Precision)

This scenario tests the system's ability to handle queries where exact-match precision is critical.

- Engineer Query: "Find issues in the 'Project1' project where ERR_STATUS_REG is 0xDEADBEEF."
- System Action: The LLM Orchestrator identifies the project as a filter and the register/value pair as high-importance keywords. The RAG module's query heavily relies on the BM25 text search and the keyword filter.
- Result: The system successfully retrieves the one or two historical tickets that exactly match this error signature.
- Value: This demonstrates the system's superiority over a pure vector-search solution, which would be incapable of finding this precise, literal string.

2) Scenario 2: Semantic-Dominant Query

This scenario tests the system's ability to handle ambiguous, high-level conceptual queries.

- Engineer Query: "Find issues where a lock-up occurred."
- System Action: The engineer used the word "lock-up" but historical reports may have used "stuck", "hang" or "timeout." The kNN vector search component identifies the semantic similarity between these concepts.
- Result: The system retrieves historical reports documented as "system hang during boot" or "core stuck", even if the specific word "lock-up" was never used in those tickets.
- Value: This demonstrates the system's ability to overcome the limitations of traditional keyword search, providing a broad, conceptual overview of an issue.

3) Scenario 3: Hybrid Query (The Common Case)

This scenario tests the system's primary capability: blending architectural precision via metadata filtering with semantic recall of failure symptoms.

- Engineer Query: "Find lock-up issues caused by L2 cache uncorrectable ECC errors in the CPU of project1"
- System Action: The LLM Orchestrator parses the intent into a structured request. The system applies strict metadata filters for project: 'project1' and block: 'CPU'. Within this scoped search space, the hybrid engine simultaneously executes a BM25 search for the keywords "L2 cache" and "UECC" and a kNN vector search for the term "lock-up".
- Result: The system returns a highly relevant, ranked list of documents specific to the CPU block of project1. It successfully surfaces a critical historical case titled "Fatal hang caused by L2 Cache double-bit error", precisely matching the hardware context and failure type.
- Value: This illustrates the best-of-both-worlds relevance, maintaining surgical precision for specific projects and IP blocks while remaining flexible enough to capture diverse documentation styles for hardware failures and component-level errors.

Qualitative Feedback:

While the illustrative retrieval scenarios highlight the system's ability to handle different query types in a controlled manner, we also collected qualitative feedback from a focus group of 10 engineers who interacted with the system using historical debug cases to assess its potential effectiveness in real-world debug workflows. Overall, the feedback has been positive and corroborates the value demonstrated in the scenarios:

- Time-to-Debug Reduction: For recurring issues where similar patterns existed in the historical data, engineers anticipated a significant reduction in time-to-debug, noting that the system could have quickly surfaced relevant historical precedents.

- Knowledge Democratization: Junior engineers reported that they were able to independently resolve complex issues by querying the system and receiving context from past debug sessions, reducing reliance on senior experts and accelerating their learning process.

- Workflow Efficiency: The automated Jira integration and context-linking received positive feedback for reducing the manual documentation effort associated with tracking and updating issues.

C. Quantitative Efficiency Analysis: Impact of Proactive Parsing

To quantify the operational impact of the framework, we measured the "Time-to-First-Insight," defined as the latency between an engineer deciding to investigate a scandump and having the parsed data ready for analysis. We compared the traditional On-Demand workflow against our framework's Proactive mode.

In the traditional workflow, a synchronous blocking time of approximately 10 minutes is incurred per scandump, as the engineer must manually execute the parsing script. Furthermore, in a multi-team debug scenario (e.g., handover from CPU to Bus team), this cost is often duplicated as each new assignee re-runs the process. In contrast, our Scandump Parsing Agent operates asynchronously. By the time an engineer opens a Jira ticket, the agent has already detected the file and completed the parsing in the background. Consequently, the engineer experiences a near-instantaneous retrieval time of approximately 5 seconds. The proactive execution model shifts the latency from the critical path of the engineer's workflow to an asynchronous background process, resulting in a 99.2% reduction in perceived wait time. Additionally, the Parse Once, Use Many paradigm eliminates redundant compute cycles. Table I summarizes these efficiency gains.

TABLE I
EFFICIENCY COMPARISON: TRADITIONAL VS. PROPOSED FRAMEWORK

Metric	Traditional Workflow (On-Demand)	Proposed Framework (Proactive)	Improvement
Parsing Model	Synchronous / Manual	Asynchronous / Automated	-
Actual Compute Time	~10 minutes	~10 minutes	(Same)
User Wait Time (Latency)	~10 minutes	< 5 seconds	~99.2% Reduction
Redundancy (3 Engineers)	30 minutes (10m x 3)	10 minutes (10m x 1)	66% Compute Saving
Data Accessibility	Local / Siloed	Centralized / Shared	-

D. Search Accuracy Benchmark

To validate the effectiveness of the proposed hybrid retrieval strategy, we conducted a benchmark comparing three retrieval methods: Keyword-only (BM25), Vector-only (Embedding), and our Hybrid approach. We curated a test set of 50 representative queries derived from actual debug scenarios and measured Recall@5—the rate at which the correct historical solution appeared within the top 5 results.

The test queries were categorized into three distinct types:

1) Keyword Queries: Focused on specific register names or hexadecimal error codes (e.g., "ERR_STATUS_REG = 0xDEADBEEF")

2) Semantic Queries: Natural language descriptions of symptoms without specific identifiers (e.g., "Find issues where a lock-up occurred.")

3) Hybrid Queries: A combination of hardware context and behavioral symptoms (e.g., "Find lock-up issues caused by L2 cache uncorrectable ECC errors in CPU of project1").

As presented in Table II, single-modality searches exhibit inherent limitations in specific hardware debug contexts. Vector search performed poorly on Keyword Queries (42%) due to the difficulty of embeddings in distinguishing precise alphanumeric identifiers. Conversely, Keyword search struggled with Semantic Queries (34%) due to vocabulary mismatch. However, the Hybrid search demonstrated robustness across all categories. It effectively preserved the peak performance of BM25 for Keyword Queries (98%) and closely matched the performance of Vector search for Semantic Queries (78%). Most importantly, it significantly outperformed both individual methods in the Hybrid Queries (88%), which represents the most frequent and complex real-world debug scenarios.

TABLE II
RETRIEVAL ACCURACY COMPARISON (RECALL@5)

Query Category	Keyword Search (BM25)	Vector Search (Embedding)	Hybrid Search (Proposed)
Keyword	98%	42%	98%
Semantic	34%	80%	78%
Hybrid	60%	66%	88%
Overall Average	64%	62.7%	88%

V. Discussion and Future Work

Preliminary experiments indicate that the proposed framework has the potential to streamline post-silicon debugging by structuring data and enabling context-aware retrieval. Although full validation is still in progress, the initial deployment demonstrated that knowledge can be systematically accumulated and queried across different stages of silicon bring-up and analysis.

Nevertheless, several limitations remain. The current Scandump Parsing Agent is highly dependent on the quality and maintenance of legacy parsing scripts, which may lead to incomplete or inconsistent summaries—a typical "garbage in, garbage out" limitation. Moreover, the effectiveness of the RAG component is constrained by the

completeness and quality of the indexed data, suggesting a need for more automated and intelligent data preprocessing.

Future work will focus on the following areas:

1. Intelligent Parsing:
   We plan to investigate the use of LLM and other machine learning techniques to parse raw scandump data directly. This direction aims to reduce reliance on legacy scripts and to enable proactive anomaly detection by identifying atypical register values without explicit parsing rules.
2. Multi-Modal RAG:
   The current RAG system will be extended to incorporate diverse data sources, such as kernel logs, trace outputs, and waveform data (e.g., FSDB/VCD). Providing the LLM with richer multimodal context may enhance its ability to generate more accurate and interpretable insights.
3. Extensibility to Pre-Silicon Environments:
   A significant future direction is to apply this framework to pre-silicon verification environments. The current architecture operates on a standardized summary artifact, which provides a notable advantage for straightforward extensibility. Simulation-based environments, such as UVM testbenches, could be instrumented to produce equivalent summary outputs (e.g., summary.json) upon test completion or failure.
   This extension would provide two major benefits:
   - Unified Knowledge Infrastructure: A common Elasticsearch backend could index and manage debug summaries from both simulation and silicon runs, allowing consistent data representation across different verification stages.
   - Cross-Lifecycle Debug Capability: Engineers could perform correlated searches across pre-silicon and post-silicon data (e.g., "This UVM test failed. Has this failure signature ever been seen on real silicon?").

   By standardizing the data interface, this approach could effectively bridge the gap between pre-silicon and post-silicon debug, facilitating knowledge sharing and reducing redundant debug efforts across verification phases.

VI. Conclusion

This paper introduced a novel conversational AI agent framework designed to automate and scale the critical post-silicon debug workflow. We proposed a system where a LLM orchestrates specialized agents to unify data parsing, historical knowledge retrieval, and workflow integration. At the core of this system is a hybrid-search Retrieval-Augmented Generation (RAG) module built on Elasticsearch, designed to structure and query vast, heterogeneous debug data.

The central contribution of this work is the design and application of this Elasticsearch-based hybrid search RAG architecture, which is uniquely suited to the dual nature of hardware debug data. We demonstrated that debug analysis requires both the semantic understanding of vector search and the high-precision of keyword search. Our framework effectively unifies these modalities within a single, operationally simple platform, overcoming the limitations of pure vector-search solutions.

By applying this core RAG engine, our framework directly addresses a primary bottleneck: the repetitive, manual parsing of scandumps. It transforms siloed script outputs into a queryable, reusable knowledge base. This shifts the engineer's role from a manual data processor to a high-level analyst, significantly reducing time-to-debug. As our evaluation demonstrated, this approach offers a practical and scalable path for deployment. Looking forward, the architecture's potential to extend into pre-silicon environments suggests a promising direction for a unified, knowledge-driven debug platform across the entire semiconductor lifecycle.

References

[1] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-Silicon Validation: Opportunities, Challenges and Recent Advances," *Proc. Design Automation Conference (DAC)*, 2010, pp. 1-6

[2] D. Pal, A. Sharma, S. Ray, F. M. de Paula, and S. Vasudevan, "Application Level Hardware Tracing for Scaling Post-Silicon Debug," *Proc. Design Automation Conference (DAC)*, 2018, pp. 1-6.

[3] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *in Proc. NeurIPS*, 2020.

[4] Elasticsearch B.V., "Elasticsearch: The Official Distributed Search and Analytics Engine," [Online]. Available: https://www.elastic.co/elasticsearch/

[5] N. Wu, Y. Li, H. Yang, H. Chen, S. Dai, C. Hao, C. Yu, and Y. Xie, "Survey of Machine Learning for Software-assisted Hardware Design Verification: Past, Present, and Prospect," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 4, pp. 1-42, 2024.

[6] Milvus Project, "Milvus: An Open-Source Vector Database for Scalable Similarity Search," [Online]. Available: https://milvus.io/

[7] Pinecone Systems Inc., "Pinecone: A Fully Managed Vector Database," [Online]. Available: https://www.pinecone.io/

[8] T. Cover and P. Hart, "Nearest Neighbor Pattern Classification," IEEE Transactions on Information Theory, vol. 13, no. 1, pp. 21–27, 1967.

[9] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford, "Okapi at TREC-3," in Proceedings of the Third Text REtrieval Conference (TREC-3), 1994, pp. 109-126.
[10] LangChain Inc., "LangGraph: Multi-agent Orchestration for LLMs," [Online]. Available: https://github.com/langchain-ai/langgraph
[11] J. Chen, Y. Guo, J. Li, and K. Xu, “Optimizing Retrieval-Augmented Generation with Elasticsearch for Enhanced Question-Answering Systems,” *arXiv preprint* arXiv:2410.14167, 2024.