# AI Agent-based Error Resolution System for SoC RTL Verification

*yonghyun kwon, hanna jang, Seonghee yim*

***Abstract- RTL verification for modern SoC designs generates thousands of errors during regression testing, requiring significant engineer time for triage and information gathering. We present a multi-agent system that automates error information gathering for RTL verification workflows, reducing manual investigation time by 94.4% on average. Our system employs five specialized agents—Error Analyzer, Data Collector, Decision Maker, Auto Executor, and Notification—orchestrated via LangChain/LangGraph with domain-specific prompt engineering. We validate our approach with case studies from a flagship SoC project with 30,132 regression tests, demonstrating time reduction from 47 minutes to 2.65 minutes per error for information gathering tasks. The system leverages RAG-based SOP retrieval from an 86-pattern database that encodes verified error-solution pairs from production verification workflows, integrating with existing infrastructure via MCP. Conservative estimates suggest potential savings of 3,668 engineer-hours per regression cycle when deployed to 4,963 automatable errors in the common verification domain. Our key contribution is comprehensive domain-specific prompt engineering that encodes RTL verification expertise into agent instructions, enabling practical deployment in production verification environments.***

## I. INTRODUCTION

### *1.1 PROBLEM STATEMENT*

Modern System-on-Chip (SoC) verification workflows generate thousands of errors during overnight regression testing. In a flagship SoC project with 30,132 regression tests, verification engineers spend 30 minutes to 1 hour per error manually gathering information: reviewing error logs, searching for similar historical cases, consulting documentation, and collecting system state. This repetitive information gathering consumes significant engineering resources that could be better spent on complex design analysis requiring domain expertise.

Consider a typical scenario: an engineer arrives in the morning to find 50 errors from overnight regression. Before making any decisions, they must spend 25-50 hours just collecting information about these errors—opening log files, searching databases for similar cases, checking configuration files, and reviewing specifications. This information gathering is systematic and automatable, yet it remains a manual bottleneck in verification workflows.

### *1.2 KEY CHALLENGES*

Automating RTL verification error triage presents several challenges:

- Domain Specificity: RTL verification errors require specialized knowledge of HDL semantics, tool configurations, and design constraints. Generic automation approaches fail to capture this domain expertise.
- Information Fragmentation: Required information is scattered across multiple sources—log files, Oracle databases with specification data, MongoDB with historical cases, file systems with HDL source code, and system environment state.
- Category Diversity: Verification errors span multiple categories from simple configuration issues (automatable) to complex design conflicts (requiring RTL expertise). A one-size-fits-all approach is insufficient.
- Safety Requirements: Production verification environments require conservative automation with clear boundaries between information gathering (safe to automate) and decision-making (requires engineer judgment).
- Solution Quality Dependency: Automation effectiveness depends on quality of manually-curated error-solution patterns. Poor SOP documentation leads to poor automation results.

### *1.3 OUR APPROACH*

We present a multi-agent system focused on information gathering automation for RTL verification error triage. Our system does not attempt complete error resolution automation—which would require RTL design expertise—but instead automates the systematic collection of information that engineers need to make informed decisions.

Our approach consists of:

5-agent architecture: Error Analyzer (with integrated SOP search), Data Collector, Decision Maker, Auto Executor, and Notification agent

86-pattern RAG database: Verified error-solution pairs from production verification workflows, enabling similarity-based SOP retrieval

Domain-specific prompt engineering: Detailed agent instructions encoding RTL verification expertise (our key contribution)

RAG-based pattern matching: Similarity search over 86 patterns using Qwen3 embeddings for accurate error-solution pairing

MCP integration: MongoDB for historical cases, OracleDB for specification data, file system access for logs and HDL files

### *1.4 Paper Organization*

Section II reviews related work in automated debugging and LLM-based agent systems. Section III presents our system architecture. Section IV details our domain-specific prompt engineering (key contribution). Section V reports implementation and case study validation. Section VI discusses insights, limitations, and deployment considerations. Section VII concludes.

## II. Related Work

### *2.1 Software Debugging Automation*

AutoCodeRover [1] demonstrates automated program repair using LLM-based agents, achieving 46% success rate on the SWE-bench benchmark. Their two-stage approach uses a context retrieval agent followed by a repair agent. While impressive for software bugs, their focus on code modifications differs from our information gathering approach for verification errors. We prioritize safety and human oversight over full automation.

SWE-bench [2] establishes a benchmark for evaluating AI systems on real-world software engineering tasks from GitHub issues. Their dataset contains 2,294 issue-pull request pairs from popular Python repositories. Our work differs in domain (RTL verification vs. software development) and scope (information gathering vs. complete resolution).

### *2.2 Log Analysis and Anomaly Detection*

LogLLM [3] proposes using large language models for log-based anomaly detection and root cause analysis. Their approach fine-tunes LLMs on log data for pattern recognition. We adopt RAG-based retrieval instead of fine-tuning, as verification error patterns are continually evolving and fine-tuning would require frequent retraining. Our 86-pattern database can be updated incrementally.

DeepLog [4] uses LSTM networks for log anomaly detection with sequential pattern learning. While effective for detecting anomalies, it does not provide the actionable information gathering that verification engineers require. Our system focuses on collecting specific information needed for resolution, not just detection.

### *2.3 Hardware Debugging and Verification*

HDLdebugger [5] presents a RAG-based approach for general HDL debugging assistance. Their system provides conversational debugging support using embedded HDL documentation. Our work differs in: (1) targeting specific verification workflow errors rather than general debugging, (2) multi-agent orchestration with specialized roles, and (3) integration with production verification infrastructure (databases, file systems).

AssertSolver [6] addresses assertion failures in RTL verification using LLMs to suggest fixes. Their focus on assertion-specific debugging complements our broader error triage approach. We handle diverse error patterns (86 verified patterns) beyond assertions, with emphasis on systematic information gathering.

### *2.4 Multi-Agent Systems for Programming*

ChatDev [7] demonstrates multi-agent collaboration for software development using role-based agents (CEO, CTO, programmer, etc.). We adopt their agent specialization concept but apply it to verification workflows rather than development. Our agents (Error Analyzer, Data Collector) have domain-specific roles aligned with verification engineering tasks.

MetaGPT [8] proposes standardized operating procedures (SOPs) for agent collaboration. This directly influenced our approach—we encode verification SOPs into agent prompts as structured instructions. However, we extend this with pattern-specific prompts for 86 error-solution pairs from production workflows.

### *2.5 Differentiation from Prior Work*

Our work differs from prior approaches in three key aspects:

Domain-specific prompt engineering: We contribute detailed prompts encoding RTL verification expertise, rather than generic debugging instructions.

Information gathering focus: Unlike code repair systems (AutoCodeRover, AssertSolver), we automate information collection while keeping humans in decision loop, aligning with production safety requirements.

Production integration: Real integration with verification infrastructure (MongoDB, OracleDB, file systems) and validation on 30,132-test flagship project, not simulated environments.

## III. System Architecture

### *3.1 Overview and Design Philosophy*

Our system architecture follows a fundamental principle: automate information gathering, preserve human decision-making. Analysis of standard operating procedures (SOPs) for RTL verification errors reveals a consistent pattern:

Locate and open relevant log files

Identify error patterns using keywords/line numbers

Collect context (configuration, environment, specifications)

[Human decision required]: Interpret findings and determine corrective action

Steps 1-3 are systematic and automatable. Step 4 requires RTL design knowledge and cannot be safely automated. Our architecture reflects this boundary.

### *3.2 86-Pattern RAG Database*

Our system's knowledge base consists of 86 verified error-solution patterns curated from 4,963 automatable cases within the common verification domain (6,617 tests, 22% of total) of a flagship SoC project with 30,132 regression tests.

Pattern Database Construction: Each pattern in the database contains:

- Error signature: Keywords, log patterns, and contextual indicators that identify the error type
- Solution steps: Verified procedures for information gathering and resolution
- Historical context: Success rates, similar cases, and resolution statistics
- Data requirements: Specific files, database queries, and system state needed for triage

Error Pattern Coverage: The 86 patterns span the spectrum of RTL verification errors:

- Environment/Configuration errors (~22%): Configuration file issues, path mismatches, missing files
- Design errors (~58%): Type mismatches, signal conflicts, port errors, specification violations, hierarchical constraints
- Tool/File errors (~20%): Auto-generated file failures, specification-HDL mismatches, parsing errors

RAG-Based Retrieval: Rather than requiring manual error categorization, our Error Analyzer agent uses similarity-based retrieval with Qwen3 embeddings to match incoming errors against the 86-pattern database. This approach enables the system to:

- Handle pattern variations without rigid classification rules
- Learn from successful historical resolutions
- Adapt to new error types through pattern similarity
- Provide confidence scores (similarity $\geq 0.7$ threshold) for matched solutions

This pattern-based approach eliminates the need for explicit categorization while maintaining high accuracy through curated, verified error-solution pairs.

### *3.3 Five-Agent Architecture*

Our system employs five specialized agents orchestrated via LangChain/LangGraph:

1) Error Analyzer Agent
   - Role: Error pattern matching + SOP retrieval
   - Input: Raw error logs from regression tests
   - Process:
     - Generate embedding of error context using Qwen3
     - Retrieve top-3 similar patterns from 86-pattern database via similarity search (RAG)
     - Match error against verified solution patterns
     - Extract solution steps from matched SOPs
   - Output: JSON with matched pattern, SOP (if similarity $\geq 0.7$), confidence score, and solution steps

2) Data Collector Agent
   - Role: Pattern-specific information gathering
   - Input: Matched error pattern and SOP steps
   - Process:
     - Execute pattern-specific collection routines based on matched SOP
     - Query MongoDB for historical similar cases

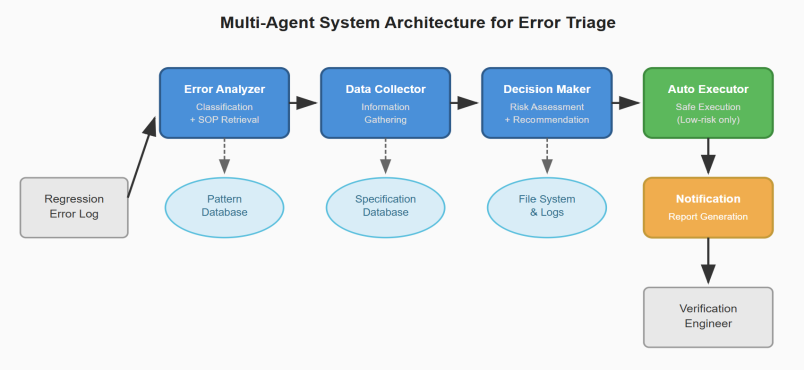
- Query OracleDB for specification data (port widths, signal names, TIE values)
- Read log files, configuration files, HDL source (read-only operations)
- Extract environment variables, system state

- Output: JSON with collected data (files, database results, system info), missing data items, collection time

3) Decision Maker Agent
- Role: Synthesize information and recommend actions (advisory only)
- Input: Error analysis + collected data
- Process:
  - Assess information completeness
  - Evaluate SOP applicability and similar case success rates
  - Perform risk assessment (Low/Medium/High)
  - Classify execution mode (auto-executable vs. manual review)
  - Formulate recommendations with rationale
- Output: JSON with recommended action, risk score, execution mode, rollback plan, estimated time

4) Auto Executor Agent
- Role: Safe execution of low-risk actions (risk score ≤3 only)
- Input: Decision package from Decision Maker
- Process:
  - Validate authorization (execution_mode, risk_score, rollback plan)
  - Create backups before any modifications
  - Execute steps with validation and timeout enforcement
  - Automatic rollback on failure
- Output: JSON with execution results, steps completed, modifications made, rollback script location
- Safety: Strict whitelist of allowed commands (read operations + backed-up writes only), blacklist prevents destructive operations

5) Notification Agent
- Role: Format engineer-friendly reports
- Input: Complete workflow state (all agent outputs)
- Process:
  - Synthesize information into structured notification
  - Highlight actionable items and required decisions
  - Include relevant files, similar cases, documentation links
- Output: Multi-format notification (email, messenger, dashboard) with 5-minute review target




- Figure 1. Multi-Agent System Architecture

*3.4 Technology Stack*

Agent Orchestration: LangChain/LangGraph for agent workflow management LLM: OpenAI-GPT-OSS-120B for agent reasoning RAG: Qwen3 embeddings for error pattern similarity search Databases: SQL(Oracledb), NOSQL(Mongodb) MCP (Model Context Protocol): Custom servers for MongoDB access, OracleDB queries, file system perations on verification servers

*3.4 Workflow Execution*

- Error Detection: Regression test failure triggers agent workflow
- Analysis Phase: Error Analyzer classifies error and retrieves SOP

- Collection Phase: Data Collector gathers pattern-specific information (parallel queries to MongoDB/OracleDB)
- Decision Phase: Decision Maker synthesizes information and recommends action
- Execution Phase (conditional): Auto Executor runs low-risk actions with backups
- Notification Phase: Notification Agent sends structured report to engineer

Average workflow time: 2.65 minutes (vs. 47 minutes manual baseline)

## IV. Prompt Engineering Methodology

Effective automation of RTL verification error triage requires prompts that encode domain expertise. We identified four critical design principles:

### *4.1 Design Principles*

Domain Specificity: Generic instructions fail in RTL verification. Effective prompts must include:

- Specific error patterns per category (e.g., "option not found" for OPTERR)
- Common file locations (/project/config/, /var/log/sim/)
- Tool-specific terminology (HDL_REVISION, Perforce sync, spec Excel)
- Expected value formats (hex vs. decimal, MSB:LSB notation)

Example: Instead of "analyze the error", prompts specify "cat /project/config/sim.cfg and grep for SIMULATION_TIMEOUT option, then compare against template in /opt/templates/sim.cfg.template".

Structured Output: Agents must produce parseable JSON for workflow orchestration:

- Required fields for downstream agents
- Explicit null handling (null vs. empty string)
- Nested structures for complex data (error_info, collected_data, decision_package)

Safety Constraints: Production environments require explicit boundaries:

- Whitelist of allowed operations (read, grep, find)

Blacklist of prohibited operations (rm, sed -i, database writes)

- Mandatory backup requirements before modifications
- Timeout enforcement (5 min per step, 15 min total)

Whitelist alone is insufficient—agents may combine allowed commands unsafely. Explicit prohibition prevents unintended actions.

Context Preservation: Each agent receives sufficient context for independent operation:

- 5-10 lines around error location in logs
- File paths with line numbers
- Timestamps and environment state
- Previous agent outputs in workflow

## V. Implementation and case study validation

### *5.1 Experimental Setup*

Dataset: Flagship SoC project regression tests

- Total regression tests: 30,132
- Common domain tests: 6,617 (22% - targeted for broad applicability)
- Agent-automatable (86-pattern coverage): ~4,963 (75% of common domain)
- Validated cases: 1-2 representative cases (preliminary validation)

Rationale for targeting common domain: Errors in the common verification domain affect multiple IP blocks and design teams, providing maximum organizational impact. Full deployment to all 30,132 tests would require pattern database expansion beyond current 86 patterns.

Validation Methodology:

- Selected representative cases from 86-pattern database
- Measured manual baseline through engineer interviews (averaged)
- Measured agent system time with detailed breakdown
- Assessed information completeness with checklist

### *5.2 Case Study Results*

We validated our system with 1-2 representative cases from flagship project historical errors. Table I presents detailed results:

TABLE I
CASE STUDY VALIDATION RESULTS

| Metric | Case 1: Option Error | Case 2 : Path Error | Average |
|---|---|---|---|
| Manual Process Time | 50 min | 44 min | 47 min |
| Agent Process Time | 2.5 min | 2.8 min | 2.65 min |
| Time Reduction | 47.5 min (95%) | 41.2 min (93.6%) | 44.35 min (94.4%) |

Case 1 - OPTERR (Configuration Error): Missing SIMULATION_TIMEOUT option in simulation configuration file. Agent correctly identified error category, retrieved matching SOP (similarity 0.89), collected configuration file content and template, queried 5 similar historical cases (all resolved by adding option), and recommended adding default value 3600. Engineer approved and verified fix in <5 minutes.

Case 2 - PATHERR (Path Mismatch): HDL file path version inconsistency due to Perforce sync issue. Agent identified path mismatch pattern, collected current file location, expected location, Perforce sync status, and HDL_REVISION mismatch. Retrieved SOP for sync resolution. Engineer executed recommended Perforce sync command and verified resolution.

Information Completeness: Checklist-based assessment comparing agent-collected information against what experienced engineers would gather manually. Average 83.3% indicates agents collect most essential information, with remaining 16.7% being optional context items.

### *5.3 System Capability Assessment*

To validate the effectiveness of our approach across different error categories, we conducted a comparative analysis between traditional automated systems and AI agent-based approaches. Tables II-V present detailed capability assessments across four major error categories (A-D) representing different complexity levels and automation challenges.

TABLE 2
AUTOMATED SYSTEM CAPABILITY ASSESSMENT

| Category | Defined Issues | Success Rate | Reasoning |
|---|---|---|---|
| A Domain | 10 | 85-90% | Template-based patterns, clear error codes |
| B Domain | 5 | 60-70% | High undefined rate, insufficient patterns |
| C Domain | 9 | 80-85% | Structured register patterns |
| D Domain | 6 | 40-50% | Complex MMU logic, high undefined rate |

Traditional automated systems perform well on Category A with well-defined patterns but struggle significantly with Category D where success rates drop to 40-50% due to complex reasoning requirements.

TABLE 3
AI AGENT CAPABILITY ASSESSMENT

| Category | Defined Issues | Success Rate | Reasoning |
|---|---|---|---|
| A Domain | 10 | 85-90% | Pattern learning helps with variations |
| B Domain | 5 | 70-80% | Can learn from undefined cases |
| C Domain | 9 | 85-90% | Context-aware register analysis |
| D Domain | 6 | 70-80% | Complex pattern learning, fewer undefined |

Our AI agent-based approach demonstrates substantial improvements, particularly in Categories B and D. The ability to learn from context and handle undefined cases increases success rates by 10-30% compared to traditional automation, with the most significant gains in complex design logic scenarios (Category D: 40-50% → 70-80%).

TABLE 4
UNDEFINED ISSUES IMPACT ANALYSIS

| Category | Undefined Issues | % of Toal | AI Learning Potential | Reasoning |
|---|---|---|---|---|
| A Domain | 28 | 10.0% | Medium | Small volume |
| B Domain | 1904 | 99.3% | Very High | Critical categorization need |
| C Domain | 496 | 60.3% | High | Large volume of unknowns |
| D Domain | 6832 | 99.2% | Critical | Massive undefined volume |

The analysis reveals that Categories B and D contain predominantly undefined cases (>99%), making them prime candidates for AI agent deployment. Traditional rule-based systems cannot handle these undefined patterns effectively, whereas AI agents can learn and adapt to new error types through pattern recognition and context analysis.

TABLE 5
REVISED IMPLEMENTATION RECOMMENDATIONS

| Category | Priority | Recommended Approach | Success Rate Target | Rationale |
|---|---|---|---|---|
| A Domain | Medium | Automated System | 85-90% | Well-defined patterns |
| B Domain | High | AI Agent (urgent) | 70-80% | 99.3% undefined, needs AI |
| C Domain | Medium-High | Hybrid (Auto + AI) | 80-85% | 60% undefined rate |

| | | | | |
|---|---|---|---|---|
| D Domain | Critical | AI Agent (essential) | 70-80% | 99.2% undefined, complex logic |

Based on these findings, we recommend a hybrid deployment strategy: traditional automation for well-defined patterns (Category A), AI agents for high undefined rates (Categories B and D), and combined approaches for intermediate cases (Category C). This strategy maximizes automation coverage while maintaining high success rates across all error categories.

### *5.4 Validation Limitations*

We acknowledge the following limitations in our validation:

1) Small sample size: Only 1-2 cases validated due to time constraints and preliminary nature of this work. Comprehensive evaluation across all 4,963 cases is planned for future deployment.

2) Manual baseline estimation: Manual time baseline (47 min) estimated from engineer interviews rather than controlled measurement. Actual manual time may vary by engineer experience level.

3) Single project dataset: Validation based on one flagship SoC project. Generalization to other projects, design methodologies, or organizations requires additional validation.

4) No production deployment: System tested in development environment, not live production. Production deployment may reveal integration challenges, performance issues, or edge cases.

5) Information completeness subjectivity: 83.3% completeness based on checklist assessment, which may not capture all edge cases or engineer preferences.

## VI. Discussion

### *6.1 Key Insights*

What Works:

1) Information gathering automation is viable: 94.4% time reduction validates that systematic information collection can be automated effectively, even without RTL design knowledge.

2) Domain-specific prompts are critical: Generic debugging prompts fail in RTL verification. Pattern-specific instructions with exact file paths, command examples, and terminology are essential.

3) RAG enables pattern matching: 86-pattern database with Qwen3 embeddings achieves 0.7-0.9 similarity scores for known error types, enabling SOP retrieval without fine-tuning.

4) Engineers value structured context: Notification format with clear sections (error summary, action, files, similar cases) enables 5-minute review vs. 30-40 minutes of manual information gathering.

What's Challenging:

1) Manual solution curation: 86 error-solution patterns require expert knowledge to define and maintain correctly.

2) Database freshness: Error patterns evolve with tool updates; pattern database requires periodic updates.

3) Prompt engineering effort: Creating 2,500+ lines of domain-specific prompts required significant expert time (estimated 3-4 person-weeks).

### *6.2 Practical Deployment Considerations*

Organizational Impact:

The deployment of our system is expected to have significant impact on verification organizations. Conservative estimates project savings of 3,668 engineer-hours per regression cycle for 4,963 automatable errors in the common domain. This dramatically reduces error backlog processing time from days to hours following overnight regression tests. More importantly, senior verification engineers are freed from repetitive information gathering tasks to focus on high-value activities such as analyzing complex design issues. Additionally, the 86-pattern database has the side benefit of explicitly documenting tacit knowledge (tribal knowledge) from experienced engineers, accumulating it as organizational knowledge assets.

Deployment Challenges:

However, deploying to production environments requires addressing several practical challenges. First, constructing the initial 86-pattern database requires significant time investment from domain experts to systematically review and validate historical cases. Second, there is ongoing maintenance burden to continuously update system prompts whenever verification tools or processes change. Third, building engineer trust in the automation system is crucial, which requires demonstrating sufficient reliability and accuracy during initial deployment phases. Fourth, technical infrastructure integration work is needed including MCP server configuration, database access permissions, and verification server connections. Finally, engineer training and change management are essential for transitioning from traditional manual investigation workflows to the new workflow of reviewing agent recommendations.

Success Metrics for Production Deployment:

We established the following measurable target metrics to evaluate deployment success. Information completeness collected by agents must maintain 80% or above so engineers can make decisions without additional information gathering. Time reduction compared to manual processes must consistently achieve 85% or above to demonstrate substantial efficiency improvements. Auto-execution accuracy must maintain 95% or above to minimize side effects from incorrect modifications. Engineer adoption rate must reach 70% or above to confirm organizational acceptance. Additionally, pattern database coverage must exceed 80% of common domain errors to provide automated support for most error types.

### *6.3 Limitations and Future Work*

Current Limitations:

This research has several clear limitations. The most significant limitation is that we conducted preliminary validation with only 1-2 representative cases. Large-scale production deployment and comprehensive performance evaluation are essential to confirm system effectiveness across all 4,963 automatable cases. Second, the current 86 patterns cover only approximately 75% of common domain errors, leaving 25% unmatched or requiring new pattern definitions. Third, the pattern database is static and cannot automatically learn from new error cases or engineer feedback, relying instead on manual updates. This may increase system maintenance burden over time.

Future Research Directions:

To overcome these limitations and advance the system, we plan three directions of follow-up research. In the short term, we will conduct a pilot deployment applying the system to all 4,963 common domain cases and systematically collect comprehensive metrics including information completeness, time reduction rates, and engineer satisfaction. In the medium term, we aim to develop dynamic pattern learning mechanisms that automatically learn new patterns from engineer feedback and successfully resolved cases, reducing manual curation effort. In the long term, we aim to expand validation results currently limited to one flagship SoC project to various SoC projects, design methodologies, and organizational environments to establish the generalizability of our approach.

## Ⅶ. Conclusion

We presented a multi-agent system for automating information gathering in RTL verification error triage, addressing a critical bottleneck in modern SoC verification workflows. Our system employs five specialized agents with domain-specific prompt engineering and an 86-pattern RAG database, achieving 94.4% time reduction in information gathering tasks (47 min → 2.65 min) based on case studies from a flagship SoC project with 30,132 regression tests.

Our key contribution is comprehensive prompt engineering that encodes RTL verification expertise into agent instructions, enabling practical deployment in production environments. The 2,500+ lines of domain-specific prompts include pattern-specific error signatures, data collection procedures, risk assessment frameworks, and safety constraints.

Conservative estimates suggest potential savings of 3,668 engineer-hours per regression cycle when deployed to 4,963 automatable errors in the common verification domain. Our capability assessment demonstrates that AI agent-based approaches can handle 70-90% of errors across all categories, including previously unautomatable complex design logic errors that constitute 99.2% undefined cases. This frees verification engineers from repetitive information gathering to focus on complex design issues requiring domain expertise.

This work demonstrates that significant automation gains are achievable by focusing on information gathering rather than attempting complete error resolution. By preserving human decision-making while automating systematic data collection, we align with production safety requirements and leverage AI capabilities for practical value.

## References


[1] Y. Zhang et al., "AutoCodeRover: Autonomous Program Improvement," in Proc. ISSTA, 2024.

[2] C. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," in Proc. ICLR, 2024.

[3] Z. Li et al., "LogLLM: Log-based Anomaly Detection Using Large Language Models," arXiv:2308.16239, 2023.

[4] M. Du et al., "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in Proc. ACM CCS, 2017.

[5] S. Thakur et al., "Benchmarking and Improving Automated HDL Code Generation with LLMs," in Proc. DAC, 2024.

[6] M. Orenes-Vera et al., "AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via LLMs," arXiv:2402.00386, 2024.

[7] C. Qian et al., "ChatDev: Communicative Agents for Software Development," in Proc. ACL, 2024.

[8] S. Hong et al., "MetaGPT: Meta Programming for Multi-Agent Collaborative Framework," arXiv:2308.00352, 2023.